Towards Instance Optimal Join Algorithms for Data in Indexes

Hung Q. Ngo Dung T. Nguyen Christopher Ré Atri Rudra

ABSTRACT

Efficient join processing has been a core algorithmic challenge in relational databases for the better part of four decades. Recently Ngo, Porat, Ré, and Rudra (PODS 2012) established join algorithms that have optimal running time for worst-case inputs. Worst-case measures can be misleading for some (or even the vast majority of) inputs. Instead, one would hope for instance optimality, e.g., an algorithm which is within some factor on every instance. In this work, we describe instance optimal join algorithms for acyclic queries (within polylog factors) when the data are stored as binary search trees. This result sheds new light on the complexity of the well-studied problem of evaluating acyclic join queries.

We also devise a novel join algorithm over higher dimensional index structures (dyadic trees) that may be exponentially more efficient than any join algorithm that uses only binary search trees. Further, we describe a pair of lower bound results that establish the following (1) Assuming the well-known 3SUM conjecture, our new index gives optimal runtime for certain class of queries. (2) Using a novel, unconditional lower bound, i.e., that does not use unproven assumptions like P $\neq$ NP, we show that no algorithm can use dyadic trees to perform bow-tie joins better than $\operatorname{poly}\log$ factors.

1. INTRODUCTION

Efficient join processing has been a core algorithmic challenge in relational databases for the better part of four decades and is related to problems in constraint programming, artificial intelligence, discrete geometry, and model theory. Recently, some of the authors of this paper (with Porat) devised an algorithm with a running time that is worst-case optimal (in data complexity) [14]; we refer to this algorithm as NPRR. Worst-case analysis gives valuable theoretical insight into the running time of algorithms, but its conclusions may be overly pessimistic. This latter belief is not new and researchers have focused on ways to get better "per-instance" results.

The gold standard result is instance optimality. Traditionally, such a result means that one proves a bound that is linear in the input *and* output size for *every* instance (ignoring polylog factors). This was, in fact, obtained for acyclic natural join queries by Yannakakis' classic algorithm [21]. However, we contend that this scenario may not accurately measure optimality for database query algorithms. In particular, in the result above the runtime includes the time to process the input. However, in database systems, data is often pre-processed into indexes after which many queries are run using the same indexes. In such a scenario, it may make more sense to ignore the offline pre-processing cost, which is amortized over several queries. Instead, we might want to consider only the online cost of computing the join query given the indexes. This raises the intriguing possibility that one might have *sub-linear*-time algorithms to compute queries. Consider the following example that shows how a little bit of precomputation (sorting) can change the algorithmic landscape:

EXAMPLE 1.1. *Suppose one is given two sequences of integers $A = \{a_i\}_{i=1}^N$ such that $a_1 \leq \cdots \leq a_N$ and $B = \{b_j\}_{j=1}^N$ such that $b_1 \leq \cdots \leq b_N$. The goal is to construct the intersection of A and B, efficiently.*

Consider the case when $a_i = 2i$ and $b_j = 2j+1$. The intersection is disjoint, but any algorithm seems to need to ping-pong back and forth between A and B. Indeed, one can show that any algorithm needs $\Omega(N)$ time.

But what if $a_N < b_1$? In this case, $A \cap B = \emptyset$ again, but the following algorithm runs in time $\Theta(\log N)$: skip to the end of the first list, see that the intersection is empty, and then continue. This simple algorithm is essentially optimal for this instance (see Sec. 2.2 for a precise statement).

Worst-case analysis is not sensitive enough to detect the difference between the two examples above—a worst-case optimal algorithm could run in time $\Omega(N)$ on *all* intersections of size N and still be worst-case optimal. Further, note that the traditional instance optimal run time would also be $\Omega(N)$ in both cases. Thus, both such algorithms may be *exponentially slower* than an instance optimal algorithm on some instances (such algorithms run in time N, while the optimal takes only $\log N$ time).

In this work, we discover some settings where one can develop join algorithms that are instance optimal (up to polylog factors). In particular, we present such an algorithm for acyclic queries assuming data is stored in Binary Search Trees (henceforth BSTs), which may now run in sublinear time. Our second contribution is to show that using more sophisticated (yet natural and well-studied) indexes may result in instance optimal algorithms for some acyclic queries that are exponentially better than our first instance optimal algorithm (for BSTs).

Our technical development starts with an observation made by Melhorn [13] and used more recently by Demaine, López-Ortiz, and Munro [7] (henceforth DLM) about efficiently intersecting sorted lists. DLM describes a simple algorithm that allows one to *adapt* to the instance, which they show is instance optimal.[1]

One of DLM's ideas that we use in this work is how to derive a lower bound on the running time of any algorithm. Any algorithm for the intersection problem must, of course, generate the intersection output. In addition, any such algorithm must also *prove (perhaps implicitly) that any element that the algorithm does not emit is not part of the output.* In DLM's work and ours the format of such a proof is a set of propositional statements that make comparisons between elements of the input. For example, a proof may say $a_5 < b_7$ which is interpreted as saying, "*the fifth element of A* (a_5) *is smaller than seventh element of B* (b_7)" or "a_3 *and* b_8 *are equal.*" The proof is valid in the sense that any instance that satisfies such a proof must have exactly the same intersection. DLM reasons about the size of this proof to derive lower bounds on the running time of any algorithm. We also use this technique in our work.

Efficient list intersection and efficient join processing are intimately related. For example, $R(A) \bowtie S(A)$ computes the intersection between two sets that are encoded as relations. Our first technical result is to extend DLM's result to handle hierarchical join queries, e.g.,

$$H_n = R_1(A_1) \bowtie R_2(A_1, A_2) \bowtie \cdots \bowtie R_n(A_1, \ldots, A_n)$$

when the relations are sorted in lexicographical order (BST indexes on $A_1, \ldots, A_i$ for $i = 1, \ldots, n$). Intuitively, solving H_n is equivalent to a sequence of nested intersections. For such queries, we can use DLM's ideas to develop instance optimal algorithms (up to $\log N$ factors where $N = \max_{i=1,\ldots,n} |R_i|$). There are some minor technical twists: we must be careful about how we represent intermediate results from these joins, and the book keeping is more involved than DLM's case.

Of course, not all joins are hierarchical. The simplest example of a non-hierarchical query is the bow-tie query:

$$R(A) \bowtie S(A, B) \bowtie T(B)$$

[1]This argument for two sets has been known since 1972 [12].

We first consider the case when there is a single, traditional BST index on S, say in lexicographic order A followed by B while R (resp. T) is sorted by A (resp. B). To compute the join $R(A) \bowtie S(A, B)$, we can use the hierarchical algorithm above. This process leaves us with a new problem: we have created sets indexed by different values for the attribute A, which we denote $U_a = \sigma_{A=a}(R(A) \bowtie S(A, B))$ for each $a \in A$. Our goal is to form the intersection $U_a \cap T(A)$ for each such a. This procedure performs the same intersection many times. Thus, one may wonder if it is possible to cleverly arrange these intersections to reduce the overall running time. However, we show that while this clever rearrangement can happen, it affects the running time by at most a constant factor.

We then extend this result to all acyclic queries under the assumption that the indexes are *consistently ordered*, by which we mean that there exists a total order on all attributes and the keys for the index for each relation are consistent with that order. Further, we assume the order of the attributes is also a reverse elimination order (REO), i.e., the order in which Yannakakis processes the query (For completeness, we recall the definition in Appendix D.5.2). There are two ideas to handle such queries: (1) we must proceed in round-robin manner through the joins between several joins between pairs of relations. We use this to argue that our algorithm generates at least one comparison that subsumes a unique comparison from the optimal proof in each iteration. And, (2) we must be able to efficiently infer which tuples should be omitted from the output from the proof that we have generated during execution. Here, by efficient we mean that each inference can be performed in time poly log in the size of the data (and so in the size of the proof generated so far). These two statements allow us to show that our proposed algorithm is optimal to within a poly log factor that depends only on the query size. There are many delicate details that we need to handle to implement these two statements. (See Section 3.3 for more details.)

We describe instances where our algorithm uses binary trees to run exponentially faster than previous approaches. We show that the runtime of our algorithm is never worse than Yannakakis' algorithm for acyclic join queries. We also show how to incorporate our algorithm into NPRR to speed up acyclic join processing for certain class of instances, while retaining its worst-case guarantee. We show in Appendix G that the resulting algorithm may also be faster than the recently proposed Leapfrog-join that improved and simplified NPRR [19].

Beyond BSTs. All of the above results use binary search trees to index the data. While these data structures are ubiquitous in modern database systems, from a theoretical perspective they may not be optimal for join pro-

cessing. This line of thought leads to the second set of results in our paper: *Is there a pair of index structure and algorithm that allows one to execute the bow-tie query more efficiently?*

We devise a novel algorithm that uses a common, index structure, a *dyadic tree* (or 2D-BST), that admits 2D rectangular range queries [2]. The main idea is to use this index to support a lazy book keeping strategy that intuitively tracks "where to probe next." We show that this algorithm can perform exponentially better than approaches using traditional BSTs. We characterize an instance by the complexity of encoding the "holes" in the instance which measure roughly how many different items we have to prune along each axis. We show that our algorithm runs in time quadratic in the number of holes. It is straightforward from our results to establish that no algorithm can run faster than linear in the number of holes. But this lower bound leaves a potential quadratic gap. Assuming a widely believed conjecture in computational geometry (the 3SUM conjecture [17]), we are able to show an algorithm that is faster than quadratic in the number of holes is unlikely. We view these results as a first step toward stronger notions of optimality for join processing.

We then ask a slightly refined question: *can one use the 2D-BST index structure to perform joins substantially faster?* Assuming the 3SUM conjecture, the answer is no. However, this is not the best one could hope for as 3SUM is an unproven conjecture. Instead, we demonstrate a geometric lower bound that is unconditional in that the lower bound does not rely on such unproven conjectures. Thus, our algorithm uses the index optimally. We then extend this result by showing matching upper and (unconditional lower bounds) for higher-arity analogs of the bow-tie query.

2. BACKGROUND

We give background on binary-search trees in one and two dimensions to define our notation. We then give a short background about the list intersection problem (our notation here follows DLM).

2.1 Binary Search Trees

In this section, we recap the definition of (1D and) 2D-BST and record some of their properties that will be useful for us.

One-Dimensional BST. We begin with some properties of the one-dimensional BST, which would be useful later. Given a set U with N elements, the 1D-BST for U is a balanced binary tree with N leaves arranged in increasing order from left to right. Alternatively, let r be the root of the 1D-BST for U. Then the subtree rooted at the left child of r contains the $\lfloor \frac{N}{2} \rfloor$ smallest elements from U and the subtree rooted at the right child of r contains the $\lceil \frac{N}{2} \rceil$ largest elements in U. The rest of the tree is defined in a similar recursive manner. For a given tree $\mathcal{T}$ and a node v in $\mathcal{T}$, let $\mathcal{T}_v$ denote the subtree of $\mathcal{T}$ rooted at v. Further, at each node v in the tree, we will maintain the smallest and largest numbers in the sub-tree rooted at it (and will denote them by ℓ_v and r_v respectively). Finally, at node v, we will store the value $n_v = |\mathcal{T}_v|$.[2]

The following claim is easy to see:

PROPOSITION 2.1. *The 1D-BST for N numbers can be computed in $O(N \log N)$ time.*

LEMMA 2.2. *Given any BST $\mathcal{T}$ for the set U and any interval $[\ell, r]$ one can represent $[\ell, r] \cap U$ with subset W of vertices of $\mathcal{T}$ of size $|W| \leq O(\log |U|)$ such that the intersection is at the leaves of the forest $\cup_{v \in W} \mathcal{T}_v$. Further, this set can be computed in $O(\log |U|)$ time.*

REMARK 2.3. *The proof of Lemma 2.2 also implies that all intervals are disjoint. Further, the vertices are added to W in the sorted order of their ℓ (and hence, r) values.*

For future use, we record a notation:

DEFINITION 2.4. *Given an interval I and a BST for $\mathcal{T}$, we use $W(I, \mathcal{T})$ to denote the W as defined in Lemma 2.2.*

We will need the following lemma in our final result:

LEMMA 2.5. *Let $\mathcal{T}$ be a 1D-BST for the set U and consider two intervals $I_1 \supseteq I_2$. Further, define $U_{1 \setminus 2} = (I_1 \setminus I_2) \cap U$. Then one can traverse the leaves in $\mathcal{T}$ corresponding to $U_{1 \setminus 2}$ (and identify them) in time*

$$O\left(\left(|U_{1\setminus 2}| + |W(I_2, \mathcal{T})|\right) \cdot \log |U|\right).$$

Two-Dimensional BST. We now describe the data structure that can be used to compute range queries on 2D data. Let us assume that U is a set of n pairs (x, y) of integers. The 2D-BST $\mathcal{T}$ is computed as follows.

Let $\mathcal{T}^X$ denote the BST on the x values of the points. vertex v, we will denote the interval of v in $\mathcal{T}^X$ as $[\ell_v^x, r_v^x]$. Then for every vertex v in $\mathcal{T}^X$, we have a BST (denoted by $\mathcal{T}^Y(v)$) on the y values such that $(x, y) \in U$ and x appears on a leaf of $\mathcal{T}_v^X$ (i.e. $x \in [\ell_v, r_v]$). If the same y value appears for more than one x such that $x \in [\ell_v, r_v]$, then we also store the number of such y's on the leaves (and compute n_v for the internal nodes so that it is the weighted sum of the values on the leaves).

For example, consider the set U in Figure 1. Its 2D-BST is illustrated in Figure 4.

We record the following simple lemma that follows immediately from Lemma 2.2.

[2]If the leaves are weighted then n_v will be the sum of the weights of all leaves in $\mathcal{T}_v$.

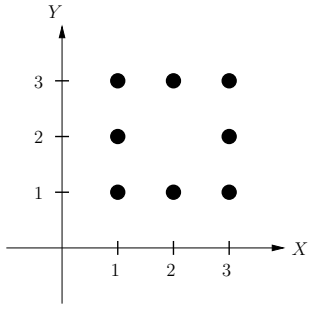

Figure 1: A set $U = [3] \times [3] - \{(2,2)\}$ of eight points in two dimension.

LEMMA 2.6. *Let v be a vertex in $\mathcal{T}^X$. Then given any interval I on the y values, one can compute whether there is any leaf in $\mathcal{T}^Y(v)$ with value in I (as well as get a description of the intersection) in $O(\log N)$ time.*

2.2 List Intersection Problem

Given a collection of of n sets $A_1, \ldots, A_n$, each presented in sorted order as follows:

$$A_s = \{A_s[1], \ldots, A_s[N_s]\} \text{ where } A_s[i] < A_s[j]$$

for all s and $i < j$. We want to output the intersection of n sets A_i, $i = 1, 2, \ldots, n$.

To do that, DLM introduced the notion of an argument.

DEFINITION 2.7. *An* argument *is a finite set of symbolic equalities and inequalities, or* comparisons*, of the following forms: (1) $(A_s[i] < A_t[j])$ or (2) $A_s[i] = A_t[j]$ for $i, j \geq 1$ and $s, t \in [n]$. An instance satisfies an argument if all the comparisons in the argument hold for that instance.*

Some arguments define their output (up to isomorphism). Such arguments are interesting to us:

DEFINITION 2.8. *An argument P is called a B-proof if any collection of sets $A_1, \ldots, A_n$ that satisfy P, we have $\bigcap_{i=1}^n A_i = B$, i.e., the intersection is exactly B.*

LEMMA 2.9. *An argument P is a B-proof for the intersection problem precisely if there are elements b_1, ..., b_n for each $b \in B$, where b_i is an element of A_i and has the same value as b, such that*

- *for each $b \in B$, there is a tree on n vertices, every edge (i, j) of which satisfies $(b_i = b_j) \in P$; and*
- *for consecutive values $b, c \in B \cup \{+\infty, -\infty\}$, the subargument involving the following elements is a $\emptyset$-proof for that subinstance: from each A_i, take the elements strictly between b_i and c_i.*

Algorithm 1 Fewest-Comparisons For Sets

Input: A_i in sorted order for $i = 1, \ldots, n$.
Output: The a smallest B-Proof where $B = \cap_{i=1}^n A_i$

```
1:  e ← max_{i=1,...,n} A_i[1].
2:  While not done do
3:      Let e_i be the largest value in A_i such that e_i < e
4:      Let e'_i be e_i's immediate successor in A_i.
5:      If e'_j does not exist break (done)
6:      Let i_0 = argmax_{i=1,...,n} e'_i.
7:      If e = e'_i for every i = 1, ..., n then
8:          emit e'_i = e'_{i+1} for i = 1, ..., n − 1.
9:      else
10:         emit e'_{i_0} < e.
11:     e ← e'_{i_0}
```

PROOF. Suppose an argument B has those two properties in the above lemma. The first property implies that for every $b \in B$, all sets A_i also contains b. So the set B is the subset of the intersection of n sets A_i, $1 \leq i \leq n$. The second property implies that for any consecutive values $b, c \in B \cup \{+\infty, -\infty\}$, there exists no value x strictly between b and c such that all sets A_i contains x. In other words, the intersection of n sets A_i is the subset of B. So the argument P is a B-proof. □

It is not necessary that every argument P that is a B-proof has the 2 properties above. However, for any intersection set instance, there always exists a proof that has those properties. We describe these results in Appendix B.2.

We describe how the list intersection analysis works, which we will leverage in later sections. First, we describe an algorithm, Algorithm 1, that generates the fewest possible comparisons. We will then argue that this algorithm can be implemented and run in time proportional to the size of that proof.

THEOREM 2.10. *For any given instance, Algorithm 1 generates a proof for the intersection problem with the fewest number of comparisons possible.*

PROOF. For simplicity, we will prove for the intersection problem of 2 sets A and B. The case of $n > 2$ is very similar. Without loss of generality, suppose that $A[1] < B[1]$. If $B[1] \notin A$ then define i to be the maximum number such that $A[i] < B[1]$. Then the comparison $(A[i] < B[1])$ is the largest possible index and any proof needs to include at least this inequality. This is implemented above. If $B[1] \in A$ then define i to be the index such that $A[i] = B[1]$. Then the comparison $(A[i] = B[1])$ should be included in the proof for the same reason. Inductively, we start again with the set A from $(i+1)^{th}$ element and set B from $B[1]$. Thus, the Algorithm 1 generates a proof for the intersection problem with the fewest comparisons possible. □

In Algorithm 1, there is only one line inside the while loop whose running time depends on the data set size: Line 3 requires that we search in the data set, but since set is sorted a binary search can perform this in $O(\log N)$ time where $N = \max_{i=1,\ldots,n} |A_i|$. Thus, we have shown:

COROLLARY 2.11. *Using the notation above and given sets $A_1, \ldots, A_n$ in sorted order, let $\mathcal{D}$ be the fewest number of comparisons that are needed to compute $B = \bigcap_{i=1}^{n} A_i$. Then, there is an algorithm to run in time $O(n\mathcal{D} \log N)$.*

Informally, this algorithm has a running time with optimal data complexity (up to $\log N$ factors).

3. INSTANCE OPTIMAL JOINS WITH TRADITIONAL BINARY SEARCH TREES

In this section, we consider the case when every relation is stored as a single binary search tree. We describe three results for increasingly broad classes of queries that achieve instance optimality up to a $\log N$ factor (where N is the size of the largest relation in the input). (1) A standard algorithm for what we call hierarchical queries, which are essentially nested intersections; this result is a warmup that describes the method of proof for our lower bounds and style of argument in this section. (2) We describe an algorithm for the simplest non-hierarchical query that we call bow-tie queries (and will be studied in Section 4). The key idea here is that one must be careful about representing the intermediate output size, and a result that allows us to show that solving one bow-tie query can be decomposed into several hierarchical queries with only a small blowup over the optimal proof size. (3) We describe our results for acyclic join queries; this result combines the previous two results, but has a twist: in more complex queries, there are subtle inferences made based on inequalities. We give an algorithm to perform this inference efficiently.

3.1 Warmup: Hierarchical Queries

In this section, we consider join queries that we call *hierarchical*. We begin with an example to simplify our explanation and notation. We define the following family of queries; for each $n \geq 1$ define H_n as follows

$$H_n = R_1(A_1) \bowtie R_2(A_1, A_2) \bowtie \cdots \bowtie R_n(A_1, \ldots, A_n).$$

We assume that all relations are sorted in lexicographic order by attribute. Thus, all tuples in R_i are totally ordered. We write $R_i[k]$ to denote the k^{th} tuple in R_i in order, e.g., $R_i[1]$ is the first tuple in R_i. An argument here is a set of symbolic comparisons of the form: (1) $R_s[i] \leq R_s[j]$, which means that $R_s[i]$ comes before $R_t[j]$ in dictionary order, or (2) $R_s[i] = R_t[j]$, which means that $R_s[i]$ and $R_t[j]$ agree on the first k components where $k = \min\{s, t\}$. The notion of B-proof carries over immediately.

Algorithm 2 Fewest-Comparisons For Hierarchical Queries

Input: A hierarchical query H_n
Output: A proof of the output of H_n

```
1: e = max_{i=1,...,n} R_i[1] // e is the maximum initial value.
2: While not done do
3:     let e_i be the largest tuple in A_i s.t. e_i < e
4:     let e'_i be the successor of e_i for i = 1, ..., n.
5:     If there is no such e'_j then break (done)
6:     i_0 ← argmax_{j=1,...,n} e'_j
7:     // NB: i_0 = n if {e'_i}_{i=1}^n agree on all attributes
8:     If {e'_i}_{i=1}^n agree on all attributes then
9:         emit e'_n in H_n and relevant equalities.
10:        e ← the immediate successor of e
11:    else
12:        emit e_{i_0} < e
13:        e ← e'_{i_0}.
```

Our first step is to provide an algorithm that produces a proof with the fewest number of comparisons; we denote the number of comparisons in the smallest proof as $\mathcal{D}$. This algorithm will allow us to deduce a lower bound for any algorithm. Then, we show that we can compute H_n in time $O(n\mathcal{D} \log N + |H_n|)$ in which $N = \max_{i=1,\ldots,n} |R_i|$; this running time is data complexity optimal up to $\log N$. The algorithm we use to demonstrate the lower bound argument is in Algorithm 2.

PROPOSITION 3.1. *For any given hierarchical join query instance,* Algorithm 2 *generates a proof that contains no more comparisons than the hierarchical join query problem with the fewest comparisons possible.*

PROOF. We only prove that all emissions of the algorithm are necessary. Fix an output set of H_n and call it O. At each step, the algorithm tries to set the eliminator, e, to the largest possible value. There are 2 emissions to the output: (1) We only emit each tuple in the output once, since e is advanced on each iteration. Thus, each of these emissions is necessary. (2) Suppose that all e'_i do not agree, then we need to emit some inequality constraint. Notice that $e = e'_i$ for some i and that e_{i_0} is from a different relation than e : otherwise, $e'_{i_0} = e$ – if this were true for all relations we would get a contradiction to there being some e'_i that disagrees. If we omit $e_{i_0} < e$, then we could construct an instance that agrees with our proof but allows one to set $e_{i_0} = e$. However, if we do that for all values then we could get a new output tuple since this tuple would agree on a all

attributes, and this would no longer be a O-proof. □

Observe that in Algorithm 2, in each iteration, the only operation whose execution time depends on the dataset size is in Line 3, i.e., all other operations are constant or $O(n)$ time. Since each relation is sorted, this operation takes at most $\max_i \log |A_i|$ using binary search. So we immediately have the following corollary of an efficient algorithm.

COROLLARY 3.2. *Computing $H_n = R_1 \bowtie \cdots \bowtie R_n$ of the hierarchical query problem, where every relation R_i has i attributes $A_1, \ldots, A_i$ and is sorted in that order. Denote $N = \max\{|R_1|, |R_2|, \ldots, |R_n|\}$ and $\mathcal{D}$ be the size of the minimum proof of this instance. Then H_n can be computed in time $O(n\mathcal{D} \log N + |H_n|)$.*

It is straightforward to extend this algorithm and analysis to the following class of queries:

DEFINITION 3.3. *Any query Q with a single relation is* hierarchical *and if $Q = R_1 \bowtie \cdots \bowtie R_n$ is* hierarchical *and R is any relation distinct from R_j for $j = 1, \ldots, n$ that contains all attributes of Q then $Q' = R_1 \bowtie \cdots \bowtie R_n \bowtie R$ is* hierarchical.

And one can show:

COROLLARY 3.4. *If Q is a hierarchical query on relations $R_1, \ldots, R_n$ then there is an algorithm that runs in time $O(n\mathcal{D} \log N + |Q|)$ where $N = \max_{i=1,\ldots,n} |R_i|$.*

Thus, our algorithm's run time has data complexity that is optimal to within $\log N$ factors.

3.2 One-index BST for the Bow-Tie Query

The simplest example of a non-hierarchical query, and the query that we consider in this section, we call the *bow-tie query*:

$$Q_{\bowtie} = R(X) \bowtie S(X, Y) \bowtie T(Y).$$

We consider the classical case in which there is a single, standard BST on S with keys in dictionary order. Without loss, we assume the index is ordered by X followed by Y. A straightforward way to process the bow-tie query in this setting is in two steps: (1) Compute $S'(X, Y) = R(X) \bowtie S(X, Y)$ using the algorithm for hierarchical joins in the last section (with one twist) and (2) compute $S'_{[x]}(Y) \bowtie T(Y)$ by using the intersection algorithm for each x in which $S'_{[x]} = \sigma_{X=x}(S)$. Notice that the data in S' is produced in the order X followed by Y. This algorithm is essentially the join algorithm implemented in every database modulo the small twist we describe below. In this subsection, we show that this algorithm is optimal up to a $\log N$ factor (where $N = \max\{|R|, |S|, |T|\}$).

The twist in (1) is that we do not materialize the output of S'; this is in contrast to a traditional relational database. Instead, we use the list intersection algorithm to identify those x such that would appear in the output of $R(x), S(x, y)$. Notice, the projection $\pi_X(S)$ is available in time $|\pi_X(S)| \log |S|$ time using the BST. Then, we retain only a pointer for each x into its BST, which gives us the values associated with x in sorted order.[3] This takes only time proportional to the number of matching elements in S (up to $\log |S|$ factors).

The main technical obstacle is the analysis of step (2). One can view the problem in step (2) as equivalent to the following problem: We are given a set B in sorted order (mirroring T above) and m sets $Y_1 \ldots, Y_m$. Our goal is to produce $A_i = Y_i \cap B$ for $i = 1, \ldots, m$. The technical concern is that since we are repeatedly intersecting each of the Y_i sets, we could perhaps be smarter and cleverly intersect the Y_i lists to amortize part of the computation and thereby lower the total cost of these repeated intersections. Indeed, this can happen (as we illustrate in the proof); but we demonstrate that the overall running time will change by only a factor of at most 2.

The first step is to describe an Algorithm 3 to produce a proof of the contents of A_i that has the following property: if the optimal proof is of length $\mathcal{D}$, Algorithm 3 produces a proof with $2\mathcal{D}$ comparisons. Moreover, all proofs produced by the algorithm compare only elements of Y_i (for $i = 1, \ldots, m$) with elements of B. We then argue that step (2) to produce each A_i independently runs in time $O(\mathcal{D} \log N)$. For brevity, the algorithm description in Algorithm 3 assumes that the smallest element of B is smaller than any element of Y_i for $i = 1, \ldots, m$ initially. In the appendix, we include a more complete pseudocode.

PROPOSITION 3.5. *With the notation above, if the minimal sized proof contains $\mathcal{D}$ comparisons, then Algorithm 3 emits at most $2\mathcal{D}$ comparisons between elements of B and Y_i for $i = 1, \ldots, m$*

We perform the proof in two stages in the Appendix: The first step is to describe simple algorithm to generate the actual minimal-sized proof, which we use in the second step to convert that proof to one in which all comparisons are between elements of Y_j for $j = 1, \ldots, m$ and elements B. The minimal-sized proof may make comparisons between elements of $y \in Y_i$ and $y' \in Y_j$ that allow it to be shorter than the proof generated above. For example, if we have $s < l_1 = u_1 < l_2 = u_2 < s'$ we can simply write $s < l_1$, $l_1 < l_2$, and $l_2 < s'$ with three comparisons. In contrast, Algorithm 3 would generate four inequalities: $s < l_1$, $s < l_2$, $l_1 < s'$, and

[3] Equivalently, in Line 9 of Alg. 2, we modify this to emit all tuples between e'_n and e''_n where this is the largest tuple such that agrees with e'_{n-1} and then update e accordingly. This operation can be done in time log of the gap between these tuples, which means it is sublinear in the output size.

Algorithm 3 Fewest-Comparisons 1BST

Input: A set B and m sets $Y_1, \ldots, Y_m$
Output: Proof of $B \cap Y_i$ for $i = 1, \ldots, m$

```
1:  Active = [m] // initially all sets are active.
2:  While Exists active element in B and Active ≠ ∅ do
3:      l_j ← the min element in Y_j for j ∈ Active.
4:      s ← the max element, s ≤ l_j forall j ∈ Active.
5:      s' ← be s's successor in S (if s' exists).
6:      If s' does not exist then
7:          For j ∈ Active do
8:              Emit l_j θ s for θ ∈ {<, >, =}.
9:      else
10:         u_j ← the max element in Y_j s.t. u_j ≤ s'.
11:         For j ∈ Active do
12:             Emit s θ l_j and u_j θ s' for θ ∈ {=, <}.
13:             Eliminate elements a ∈ A_j, s.t. a < u_j
14:             Remove j from Active, if necessary.
15:         Eliminate elements x ∈ B s.t. x < s'.
```

$l_2 < s$. To see that this slop is within a factor 2, one can always replace a comparison $y < y'$ with a pair of comparisons $y' \ \theta \ x'$ and $x \ \theta \ y$ for $\theta \in \{<, =\}$ where x (resp. x') is the maximum (resp. minimum) element in B less than y (resp. greater than) y'. As we argued above, the pairwise intersection algorithm runs in time $O(\mathcal{D} \log N)$, while the proof above says that any algorithm needs $\Omega(\mathcal{D})$ time. Thus, we have shown:

Corollary 3.6. *For the bow-tie query, $Q_{\bowtie}$ defined above, when each relation is stored in a single BST, there exists an algorithm that runs in time $O(n\mathcal{D} \log N + |Q|)$ in which $N = \max\{|R|, |S|, |T|\}$ and $\mathcal{D}$ is the minimum number of comparisons in any proof.*

Thus, for bow-tie queries with a single index we get instance optimal results up to poly log factors.

3.3 Instance Optimal Acyclic Queries with Reverse Elimination Order of Attributes

We consider *acyclic queries* when each relation is stored in a BST that is consistently ordered, by which we mean that the keys for the index for each relation are consistent with the reverse elimination order of attributes (REO). Acyclic queries and the REO order are defined in Abiteboul et al. [1, Ch. 6.4], and we recap these defintions in Appendix D.5.2.

In this setting, there is one additional complication (compared to $Q_{\bowtie}$) that we must handle and that we illustrate by example.

Example 3.1. *Let Q_2 join the following relations:*

$$R(X) = [N] \qquad S_1(X, X_1) = [N] \times [N]$$
$$S_2(X_1, X_2) = \{(2, 2)\} \qquad T(X_2) = \{1, 3\}$$

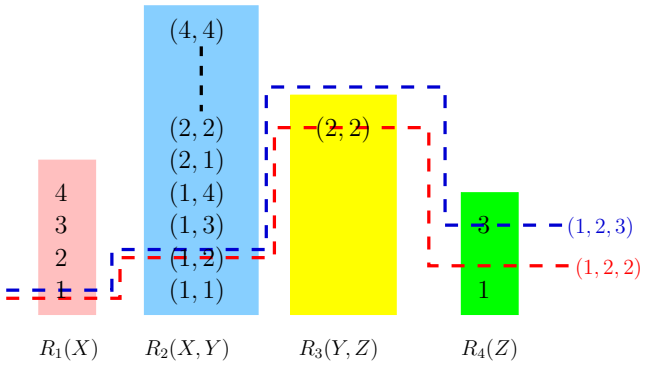

Figure 2: Illustration for the run of Algorithm 12 on the example from Example 3.1 for $N = 4$. The tuples are ordered from smallest at the bottom to largest at the top and the "probe tuple" t moves from bottom to top. The initial constraints are $X < 1$ and $X > 4$ (due to R_1), $(X, Y) < (1, 1)$ and $(X, Y) > (4, 4)$ (due to R_2), $(Y, Z) < (2, 2)$ and $(Y, Z) > (2, 2)$ (due to R_3) and $Z < 1$ and $Z > 3$ (due to R_4). Initial probe tuple t (denoted by the red dotted line) is $(1, 2, 2)$. Then we have $e_1 = e_1' = (1), e_2 = e_2' = (1, 2), e_3 = e_3' = (2, 2), e_4 = (3), e_4' = (1)$. The only new constraint added is $1 < Z < 3$. This advances the new probe tuple to $(1, 2, 3)$ and is denoted by the blue dotted line. However, at this point the constraints $(Y, Z) > (2, 2)$, $(Y, Z) < (2, 2)$ and $1 < Z < 3$ rule out all possible tuples and Algorithm 12 terminates.

The output of Q_2 is empty, and there is a short proof: $T[1].X_2 < S_2[1].X_2$ and $S_2[X].X_2 < T[1].X_2$ (this certifies that $T \bowtie S$ is empty). Naively, a DFS-style search or any join of $R \bowtie S_1$ will take $\Omega(N)$ time; thus, we need to zero in on this pair of comparisons very quickly.

In Appendix C.2, we see that running the natural modification of Algorithm 3 does discover the inequality—but it forgets it after each loop! In general, we may infer from the set of comparisons that we can safely eliminate one or more of the current tuples that we are considering. Naïvely, we could keep track of the entire proof that we have emitted so far, and on each lower bound computation ensure that takes into account all constraints. This would be expensive (as the proof may be as bigger than the input, and so the running time of this naïve approach would be least quadratic in the proof size). A more efficient approach is to build a data structure that allows us to search the proof we have emitted efficiently.

Before we talk about the data structure that lets us keep track of "ruled out" tuples, we mention the main idea behind our main algorithm in Algorithm 12. At any point of time, Algorithm 12 queries the constraint data structure, to obtain a tuple **t** that has not been

ruled out by the existing constraints. If for every $i \in [m]$, $\pi_{\text{attr}(R_i)}(\mathbf{t}) \in R_i$, then we have a valid output tuple. Otherwise, there exists a smallest $e_i > \pi_{\text{attr}(R_i)}(\mathbf{t})$ and a largest $e'_i < \pi_{\text{attr}(R_i)}(\mathbf{t})$ for some $i \in [m]$. In other words, we have found a "gap" $[e'_i + 1, e_i - 1]$. We then add this constraint to our data structure. (This is an obvious generalization of DLM algorithm for set intersection.) The main obstacle is to prove that we can charge at least one of those inserted interval to a "fresh" comparison in the optimal proof. We would like to remark that we need to generate intervals other than those of the form mentioned above to be able to do this mapping correctly. Further, unlike in the case of set intersection, we have to handle the case of comparisons between tuples of the same relation where such comparisons can dramatically shrink the size of the optimal proof. The details are deferred to the appendix.

To convert the above argument into an overall algorithm that run in time near linear in the size of the optimal proof, we need to design a data structure that is efficient. We first make the observation that we cannot hope to achieve this for any query (under standard complexity assumptions). However, we are able to show that for acyclic queries, when the attributes are ordered according to a global ordering that is consistent with an REO, then we can efficiently maintain all such prefixed constraints in a data structure that performs the inference in amortized time: $O(n2^{3n} \log N)$, which is exponential in the size of the query, but takes only $O(\log N)$ as measured by data complexity.

THEOREM 3.7. *For an acyclic query Q with the consistent ordering of attributes being the reverse elimination order (REO), one can compute its output in time*

$$O(\mathcal{D} \cdot f(n,m) \cdot \log N + mn2^{3n}|Output| \log N)$$

where $N = \max\{|R_i| \mid i = 1, \ldots, n\} + \mathcal{D}$ where $\mathcal{D}$ is the number of comparisons in the optimal proof, where $f(n,m) = mn2^{2n} + n2^{4n}$ and depends only on the size of the query and number of attributes.

A complete pseudo code for both the algorithm and data structure appears in Appendix D.

A worst-case linear-time algorithm for acyclic queries. Yannakakis' classic algorithm for acyclic queries run in time $\tilde{O}(|\text{input}| + |\text{output}|)$. Here, we ignore the small log factors and dependency on the query size. Our algorithm can actually achieve this same asymptotic runtime in the worst-case, when we do not assume that the inputs are indexed before hand. See Appendix D.2.4 for more details.

Enhancing NPRR. We can apply the above algorithm to the basic recursion structure of NPRR to speed it up considerably for a large class of input instances. Recall that in NPRR we use AGM bound [3] to estimate a subproblem size, and then decide whether to solve a subproblem before filtering the result with an existing relation. The filtering step will take linear time in the subproblem's join result. Now, we can simply run the above algorithm in parallel with NPRR and get the result of whichever finishes first. In some cases, we will be able to discover a very short proof, much shorter than the linear scan by NPRR. When the subproblems become sufficiently small, we will have an acyclic instance. In fact, in NPRR there is also a notion of consistent attribute ordering like in the above algorithm and the indices are ready-made for the above algorithm. The simplest example is when we join, say, $R[X]$ and $S[X]$. In NPRR we will have to go through each tuple in R and check (using a hash table or binary search) to see if the tuple is present in $S[X]$. If $R = [n]$ and $S = [2n] - [n]$, for example, then Algorithm 12 would have discovered that the output is empty in $\log n$ time, which is an exponential speed up over NPRR.

On the non-existence of "optimal" total order. A natural question is whether there exists a total order of attributes, depending only on the query but independent of the data, such that if each relation's BST respects the total order then the optimal proof for that instance has the least possible number of comparisons. Unfortunately the answer is no. In Appendix A we present a sample acyclic query in which, for every total order there exists a family of database instances for which the total order is infinitely worse than another total order.

4. FASTER JOINS WITH HIGHER DIMENSIONAL SEARCH TREES

This section deals with a simple question raised by our previous results: *Are there index structures that allow more efficient query processing than BST for join processing?* On some level the answer is trivially yes as one can precompute the output of a join (i.e., a materialized view). However, we are asking a more refined question: *does there exist an index structure for a single relation that allows improved join query performance?*

The answer is yes, and our approach has at its core a novel algorithm to process joins over dyadic trees. We also show a pair of lower bound results that allow us to establish the following two claims: (1) Assuming the well-known 3SUM conjecture, our new index is optimal for the bow-tie query. (2) Using a novel, unconditional lower bound[4], we show that no algorithm can use dyadic trees to perform (a generalization of) bow-tie queries up to poly log factors.

4.1 The Algorithm

[4]By unconditional, we mean that our proof does not rely on unproven conjectures like P $\neq$ NP or 3SUM hardness.

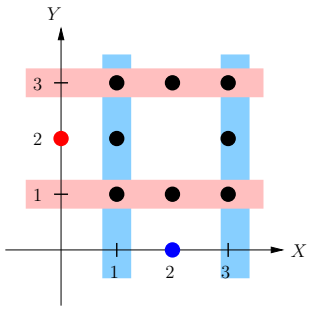

Figure 3: Holes for the case when $R = T = \{2\}$ and $S = [1,3] \times [1,3] - \{(2,2)\}$. The two X-holes are the light blue boxes and the two Y-holes are represented by the pink boxes.

Recall the *bow-tie* query, $Q_{\bowtie}$ which is defined as:

$$Q_{\bowtie} = R(X) \bowtie S(X,Y) \bowtie T(Y).$$

We assume that R and T are given to us as sorted arrays while S is given to us in a two-dimensional Binary Search Tree (2-D BST), that allows for efficient orthogonal range searches. With these data structures, we will show how to efficiently compute $Q_{\bowtie}$; in particular, we present an algorithm that is optimal on a per-instance basis for any instantiation (up to poly-log factors).

For the rest of the section we will consider the following alternate, equivalent representation of $Q_{\bowtie}$ (where we drop the explicit mention of the attributes and we think of the tables R, S and T as being input tables):

$$(R \times T) \cap S. \tag{1}$$

For notational simplicity, we will assume that $|R|, |T| \leq n$ and $|S| \leq m$ and that the domains of X and Y are integers and given two integers $\ell \leq r$, we will denote the set $\{\ell, \ldots, r\}$ by $[\ell, r]$ and the set $\{\ell+1, \ldots, r-1\}$ by (ℓ, r).

We begin with a definition of a crucial concept: *holes*, which are the higher dimensional analog of the pruning intervals in the previous section.

Definition 4.1. *We say the ith position in R (T resp.) is called an X-hole (Y-hole resp.) if there is no $(x,y) \in S$ such that $r_i < x < r_{i+1}$ ($t_i < y < t_{i+1}$ resp.), where r_j (t_j resp.) is the value in the jth position in R (T resp.) Alternatively we will call the interval (r_i, r_{i+1}) ((t_i, t_{i+1}) resp.) an X-hole (Y-hole resp.) Finally, define h_X (h_Y resp.) to be the total number of X-holes (Y-holes resp.).*

See Figure 3 for an illustration of holes for a sample bow-tie query.

Our main result for this section is the following:

Theorem 4.2. *Given an instance R, S and T of the bow-tie query as in (1) such that R and T have size at most n and are sorted in an array (or 1D-BST) and S has size m and is represented as a 2D-BST, the output $\mathcal{O}$ can be computed in time*

$$O\left(((h_X+1)\cdot(h_Y+1) + |\mathcal{O}|) \cdot \log n \cdot \log^2 m\right).$$

We will prove Theorem 4.2 in the rest of the section in stages. In particular, we will present the algorithm specialized to sub-classes of inputs so that we can introduce all the main ideas in the proof one at a time.

We begin with the simpler case where $h_Y = 0$ and the X-holes are $I_2, \ldots, I_{h_X+1}$ and we *know* all this information up front. Note that by definition, the X-holes are disjoint. Let $\mathcal{O}_X$ be the number of leaves in $\mathcal{T}^X$ such that the corresponding X values do not fall in any of the given X-holes. Thus, by Lemma 2.5 and Remark B.1 with $I_1 = (-\infty, \infty)$, in time $O((h_X + |\mathcal{O}_X|)\log m)$ we can iterate through the leaves in $\mathcal{O}_X$. Further, for each $x \in \mathcal{O}_X$, we can output all pairs $(x,y) \in S$ (let us denote this set by $\mathcal{Y}_x$) by traversing through all the leaves in $\mathcal{T}^Y(v)$, where v is the leaf corresponding to x in $\mathcal{T}^X$. This can be done in time $O(|\mathcal{Y}_x|)$. Since $h_Y = 0$, it is easy to verify that $\mathcal{O} = \cup_{x \in \mathcal{O}_X} \mathcal{Y}_x$. Finally, note that we are not exploring $\mathcal{T}^Y(u)$ for any leaf u whose corresponding x values lies in an X-hole. Overall, this implies that the total run time is $O((h_X + |\mathcal{O}|)\log m)$, which completes the proof for the special case considered at the beginning of the paragraph.

For the more general case, we will use the following lemma:

Lemma 4.3. *Given any $(x,y) \in S$, in $O(\log n)$ time one can decide which of the following hold*

(i) $x \in R$ and $y \in T$; or

(ii) $x \notin R$ (and we know the corresponding hole (ℓ_x, r_x)); or

(iii) $y \notin T$ (and we know the corresponding hole (ℓ_y, r_y)).

The proof of Lemma 4.3 as well as the rest of the proof of Theorem 4.2 are in the appendix. The final details are in Algorithm 4.

A Better Runtime Analysis. We end this section by deriving a slightly better runtime analysis of Algorithm 4 than Theorem 4.2 in Theorem 4.4 (proof sketch is in Appendix E.2). Towards that end, let $\mathcal{X}$ and $\mathcal{Y}$ denote the set of X-holes and Y-holes. Further, let $\mathcal{L}_Y$ denote the set of intervals one obtains by removing $\mathcal{Y}$ from $[y_{\min}, y_{\max}]$. (We also drop any interval from $\mathcal{L}_Y$ that does not contain any element from S.) Further, given an interval $\ell \in \mathcal{L}_Y$, let $\ell \sqcap \mathcal{X}$ denote the X-holes such that there exists at least one point in S that falls in both ℓ and the X-hole.

Theorem 4.4. *Given an instance R, S and T of the bow-tie query as in (1) such that R and T have size at*

Algorithm 4 Bow-Tie Join

Input: 2D-BST $\mathcal{T}$ for S, R and T as sorted arrays
Output: $(R \times T) \cap S$

$\mathcal{O} \leftarrow \emptyset$
Let $y_{\min}$ and $y_{\max}$ be the smallest and largest values in T
Let $\langle r \rangle$ be the state from Lemma E.1 that denotes the root node in $\mathcal{T}$
Initialize $\mathcal{L}$ be a heap with $(y_{\min}, y_{\max}, \langle r \rangle)$ with the key value being the first entry in the triple
$W \leftarrow \emptyset$
While $\mathcal{L} \neq \emptyset$ **do**
    Let (ℓ, r, P) be the smallest triple in $\mathcal{L}$
    $L \leftarrow [\ell, r]$
    While traversal on $\mathcal{T}$ for S with y values in L using Algorithm 6 is not done **do**
        Update P as per Lemma E.1
        Let (x, y) be the pair in S corresponding to the current leaf node
        Run the algorithm in Lemma 4.3 on (x, y)
        If (x, y) is in Case (i) **then**
            Add (x, y) to $\mathcal{O}$
        If (x, y) is in Case (ii) with X-hole (ℓ_x, r_x) **then**
            Compute $W([\ell_x+1, r_x-1], \mathcal{T}^X)$ using Algorithm 5
            Add $W([\ell_x + 1, r_x - 1], \mathcal{T}^X)$ to W
        If (x, y) is in Case (iii) with Y-hole (ℓ_y, r_y) **then**
            Split $L = L_1 \cup (\ell_y, r_y) \cup L_2$ from smallest to largest
            $L \leftarrow L_1$
            Add (L_2, P) into $\mathcal{L}$
Return $\mathcal{O}$

most n and are sorted in an array (or 1D-BST) and S has size m and is represented as a 2D-BST, the output $\mathcal{O}$ is computed by Algorithm 4 in time

$$O\left(\left(\sum_{\ell \in \mathcal{L}_Y} |\ell \sqcap \mathcal{X}| + |\mathcal{O}|\right) \cdot \log n \cdot \log^2 m\right).$$

We first note that since $|\mathcal{L}_Y| \leq h_Y + 1$ and $|\ell \sqcap \mathcal{X}| \leq |\mathcal{X}| = h_X$, Theorem 4.4 immediately implies Theorem 4.2. Second, we note that $\sum_{\ell \in \mathcal{L}_Y} |\ell \sqcap \mathcal{X}| + |\mathcal{O}| \leq |S|$, which then implies the following:

COROLLARY 4.5. *Algorithm 4 with parameters as in Theorem 4.2 runs in time $O(|S| \cdot \log^2 m \log n)$.*

It is natural to wonder whether the upper bound in Theorem 4.2 can be improved. Since we need to output $\mathcal{O}$, a lower bound of $\Omega(|\mathcal{O}|)$ is immediate. In Section 4.2, we show that this bound cannot be improved if we use 2D-BSTs. However, it seems plausible that one might reduce the quadratic dependence on the number of holes by potentially using a better data structure to keep track of the intersections between different holes. Next, using a result of Pătraşcu, we show that in the worst case one cannot hope to improve upon Theorem 4.2 (under a well-known assumption on the hardness of solving the 3SUM problem).

We begin with the 3SUM conjecture (we note that this conjecture pre-dates [17]– we are just using the statement from [17]):

CONJECTURE 4.6 ([17]). *In the Word RAM model with words of size $O(\log n)$ bits, any algorithm requires $n^{2-o(1)}$ time in expectation to determine whether a set $U \subset \{-n^3, \ldots, n^3\}$ of $|U| = n$ integers contains a triple of distinct $x, y, z \in U$ with $x + y = z$.*

Pătraşcu used the above conjecture to show hardness of listing triangles in certain graphs. We use the later hardness results to prove the following in Appendix E.

LEMMA 4.7. *For infinitely many integers h_X and h_Y and some constant $0 < \epsilon < 1$, if there exists an algorithm that solves every bow-tie query with h_X many X-holes and h_Y many Y-holes in time $\tilde{O}((h_X \cdot h_Y)^{1-\epsilon} + |\mathcal{O}|)$, then Conjecture 4.6 is false.*

Assuming Conjecture 4.6, our algorithm has essentially optimal run-time (i.e. we match the parameters of Theorem 4.2 up to polylog factors).

4.2 Optimal use of Higher Dimensional BSTs for Joins

We first describe a lower bound for any algorithm that uses the higher dimensional BST to process joins.

Two-dimensional case. Let $\mathcal{D}$ be a data structure that stores a set of points on the two-dimensional Euclidean plane. Let X and Y be the axes. A *box query* into $\mathcal{D}$ is a pair consisting of an X-interval and a Y-interval. The intervals can be open or close or infinite. For example, $\{[1, 5), (2, 4]\}$, $\{[1, 5], [2, 4]\}$, and $\{(-\infty, +\infty), (-\infty, 5]\}$ are all valid box queries.

The data structure $\mathcal{D}$ is called a (two-dimensional) *counting range search data structure* if it can return the number of its points that are contained in a given box query. And, $\mathcal{D}$ is called a (two-dimensional) *range search data structure* if it can return the set of all its points that are contained in a given box query. In this section, we are not concerned with the representation of the returned point set. If $\mathcal{D}$ is a dyadic 2D-BST, for example, then the returned set of points are stored in a collection of dyadic 2D-BSTs.

Let S be a set of n points on the two dimensional Euclidean plane. Let $\mathcal{X}$ be a collection of open X-intervals and $\mathcal{Y}$ be a collection of open Y-intervals. Then S is said to be *covered* by $\mathcal{X}$ and $\mathcal{Y}$ if the following holds: for each point (x, y) in S, $x \in I_x$ for some interval $I_x \in \mathcal{X}$ or $y \in I_y$ for some interval $I_y \in \mathcal{X}$, or both. We prove the following result in the appendix.

LEMMA 4.8. *Let $\mathbf{A}$ be a deterministic algorithm that verifies whether a point set S is covered by two given interval sets $\mathcal{X}$ and $\mathcal{Y}$. Suppose $\mathbf{A}$ can only access points in S via box queries to a counting range search data structure $\mathcal{D}$. Then $\mathbf{A}$ has to issue $\Omega(\min\{|\mathcal{X}| \cdot |\mathcal{Y}|, |S|\})$ box queries to $\mathcal{D}$ in the worst case.*

The above result is for the case when $\mathcal{D}$ is a counting range search data structure. We would like to prove an analogous result for the case when $\mathcal{D}$ is a range search data structure, where each box query may return a list of points in the box along with the count of the number of those points. In this case, it is not possible to show that $\mathbf{A}$ must make $\Omega(\min\{|S|, |\mathcal{X}| \cdot |\mathcal{Y}|\})$ box queries; for example, $\mathbf{A}$ can just make **one** huge box query, get all points in S, and visit each of them one by one. Fortunately, visiting the points in S takes time and our ultimate objective is to bound the run time of algorithm $\mathbf{A}$.

LEMMA 4.9. *Suppose $\mathcal{D}$ is a dyadic 2D-BST data structure that can answer box queries. Further more, along with the set of points contained in the query, suppose $\mathcal{D}$ also returns the count of the number of points in the query. Let S be the set of points in $\mathcal{D}$. Let $\mathcal{X}$ and $\mathcal{Y}$ be two collections of disjoint X-intervals and disjoint Y-intervals. Let $\mathbf{A}$ be a deterministic algorithm verifying whether S is covered by $\mathcal{X}$ and $\mathcal{Y}$, and the only way $\mathbf{A}$ can access points in S is to traverse the data structure $\mathcal{D}$. Then, $\mathbf{A}$ must run in time*

$$\Omega(\min\{|S|, |\mathcal{X}| \cdot |\mathcal{Y}|\}).$$

Now consider the bow-tie query input, where S is as defined in Lemma 4.9, R (and T resp.) consists of the end points of the intervals in $\mathcal{X}$ and $\mathcal{Y}$. Then note that checking whether $\mathcal{X}$ and $\mathcal{Y}$ cover S is equivalent to checking if the bow-tie query $R(X) \bowtie S(X, Y) \bowtie T(Y)$ is empty or not. Thus, Lemma 4.9 shows that Theorem 4.2 is tight (within poly log factors) even when $\mathcal{O} = \emptyset$.

***d*-dimensional case.** We generalize to d dimensions. First, we define the natural d dimensional version of the bow-tie query:

$$\bowtie_{i=1}^{d} R_i(X_i) \bowtie S(X_1, X_2, \ldots, X_d).$$

It is easy to check that one can generalize Algorithm 4 and thus, generalize Theorem 4.2 to compute such a query in time $O((\prod_{i=1}^{d} h_{X_i} + |\mathcal{O}|) \log^{O(d)} N)$. Next, we argue that this bound is tight if we use a d-dimensional BST to store S.

For the lower bound, consider the case where we have a point set S in $\mathbb{R}^d$, and a collection of d sets $\mathcal{X}_i$, $i \in [d]$, where for each i the set $\mathcal{X}_i$ is a set of disjoint intervals. The point set S is said to be covered by the collection $(\mathcal{X}_i)_{i=1}^{d}$ if, for every point $(x_1, \cdots, x_d) \in S$, there is an $i \in [d]$ for which x_i belongs to some interval in $\mathcal{X}_i$. We define counting range search and range search data structures in the d-dimensional case in the same way as in the Two-dimensional case. A box query Q in this case is a tuple $(I_1, \cdots, I_d)$ of d intervals, one for each coordinate $i \in [d]$. We proceed to prove the d-dimensional analog of Lemmas 4.8 and 4.9.

LEMMA 4.10. *Let $\mathbf{A}$ be a deterministic algorithm that verifies whether a point set $S \in \mathbb{R}^d$ is covered by a collection $(\mathcal{X}_i)_{i=1}^{d}$ of d interval sets. Suppose $\mathbf{A}$ can only access points in S via d-dimensional box queries to a counting range search d-dimensional data structure $\mathcal{D}$. Then $\mathbf{A}$ has to issue*

$$\Omega\left(\min\left\{\frac{1}{2^d}\prod_{i=1}^{d}|\mathcal{X}_i|, \frac{1}{4^d}|S|\right\}\right)$$

box queries to $\mathcal{D}$ in the worst case.

The proof of the following lemma is straightforward from the proof of Lemmas 4.10 and 4.9.

LEMMA 4.11. *Suppose $\mathcal{D}$ is a dyadic d-dimensional-BST data structure that can answer d-dimensional box queries. Further more, along with the set of points contained in the query, suppose $\mathcal{D}$ also returns the count of the number of points in the query. Let S be the set of points in $\mathcal{D}$. Let $\mathcal{X}_i$, $i \in [d]$, be a collection of d interval sets. Let $\mathbf{A}$ be a deterministic algorithm verifying whether S is covered by $(\mathcal{X}_i)_{i=1}^{d}$, and the only way $\mathbf{A}$ can access points in S is to traverse the data structure $\mathcal{D}$. Then, $\mathbf{A}$ must run in time*

$$\Omega\left(\min\left\{\frac{1}{2^d}\prod_{i=1}^{d}|\mathcal{X}_i|, \frac{1}{4^d}|S|\right\}\right)$$

We can easily generalize the argument after Lemma 4.9 to conclude that Lemma 4.11 implies a tight lower bound (up to polylog factors) to the upper bound on evaluating the d-dimensional bow-tie query mentioned earlier in the section.

4.3 Comparison with NPRR's Worst-case Optimal Algorithm

In this section, we first present a family of database instances for the bow-tie query where Algorithm 4 performs exponentially better than our algorithm that runs on the single index as well as the NPRR algorithm.

EXAMPLE 4.1. *Let $n \geq 3$ be an odd integer. Define $R = T = [n] \setminus \{\lfloor n/2 \rfloor, \lceil n/2 \rceil + 1\}$ and $S = [n] \times \{\lfloor n/2 \rfloor, \lceil n/2 \rceil + 1\} \cup \{\lfloor n/2 \rfloor, \lceil n/2 \rceil +\} \times [n]$. It is easy to check that the example in Figure 3 is the case of $n = 3$. Further, for every odd $n \geq 3$, we have $h_X = h_Y = 2$ and $R \bowtie S \bowtie T = \emptyset$.*

Before we talk about the run time of different algorithms on the instances in Examples 4.1, we note that we can get instance with empty output and $h_X = h_Y = 1$ (where we replace the set $\{\lfloor n/2 \rfloor, \lceil n/2 \rceil + 1\}$ by just

$\{\lfloor n/2 \rfloor\}$). However, to be consistent with our example in Figure 3 we chose the above example.

In the appendix, we show the following:

PROPOSITION 4.12. *Algorithm 4 takes $O(\log^3 n)$ on the bow-tie instances from Example 4.1, while both the NPRR algorithm and our Algorithm 3 take time $\Omega(n)$.*

In the appendix, we show that Algorithm 4 runs in time at most a poly-log factor worse than both NPRR and the Algorithm 3 on *every* instance.

PROPOSITION 4.13. *On* every *instance of a bow-tie query Algorithm 4 takes at most an $O(\log^3 n)$ factor time over NPRR and Algorithm 3.*

5. RELATED WORK

Many positive and negative results regarding conjunctive query evaluation also apply to natural join evaluation. On the negative side, both problems are NP-hard in terms of expression complexity [4], but are easy in terms of data complexity [18]. They are not fixed-parameter tractable, modulo complexity theoretic assumptions [11, 16].

On the positive side, a large class of conjunctive queries (and thus natural join queries) are tractable. In particular, the classes of acyclic queries and bounded treewidth queries can be evaluated efficiently [5,8,10,20,21]. For example, if $|q|$ is the query size, N is the input size, and Z is the output size, then Yannakakis' algorithm can evaluate acyclic natural join queries in time $\tilde{O}(\text{poly}(|q|)(N \log N + Z))$. Acyclic conjunctive queries can also be evaluated efficiently in the I/O model [15], and in the RAM model even when there are inequalities [20].

For general conjunctive queries, while the problem is intractable there are recent positive developments. A tight worst-case output size bound in terms of the input relation sizes was shown in [3]. In [14], we presented an algorithm that runs in time matching the bound, and thus it is worst-case optimal. The *leap-frog triejoin* algorithm [19] is also worst-case optimal and runs fast in practice; it is based on the idea that we can skip unmatched intervals. It is not clear how the index was built, but we believe that it is similar to our one-index case where the attribute order follows a reverse elimination order.

The problem of finding the union and intersection of two sorted arrays using the fewest number of comparisons is well-studied, dated back to at least Hwang and Lin [12] since 1972. In fact, the idea of skipping elements using a binary-search jumping (or leap-frogging) strategy was already present in [12]. Demaine et al. [7] used the leap-frogging strategy for computing the intersection of k sorted sets. They introduced the notion of "proofs" to capture the intrinsic complexity of such a problem. Then, the idea of gaps and proof encoding were introduced to show that their algorithm is average case optimal.

Geometric range searching data structures and bounds is a well-studied subject [2].[5] To the best of our knowledge the problems and lowerbounds from Lemma 4.8 to Lemma 4.11 are not known. In computational geometry, there is a large class of problems which are as hard as the 3SUM problem, and thus assuming the 3SUM conjecture there is no $o(n^2)$-algorithm to solve them [9]. Our 3SUM-hardness result in this paper adds to that list.

6. CONCLUSION AND FUTURE WORK

We have described results in two directions: (1) instance optimal results for the case when all relations are stored in BSTs where the index keys are ordered with respect to a single global order that respects a REO, and (2) we have described higher-dimensional index structures (than BSTs) to that enable instance-optimal join processing for restricted classes of queries. We showed our results are optimal in the following senses: (1) Assuming the 3SUM conjecture, our algorithms are optimal for the bow-tie query, and (2) unconditionally, our algorithm to use our index is optimal (in terms of number of probes).

We plan future work in a handful directions. First, we believe it is possible to extend our results in (1) to acyclic queries (with non-REO ordering) and cyclic queries under any globally consistent ordering of the attributes. The main idea is to enumerate not just pairwise comparisons (as we do for acyclic queries) but to enumerate all (perhaps exponentially many in the query size) paths through the query during our algorithm. We are currently working on this extension. Second, in a relational database it is often the case that there is a secondary index associated to some (or all) of the relations. While our upper-bound results still hold in this setting, our lower-bound results may not: there is the intriguing possibility that one could combine these indexes to compute the output more efficiently than our current algorithms.

We would like to point out that DLM's main results are not for instance optimality up to polylog factors; instead they consider average-case optimality up to constant factors. Such results are difficult to compare: it is a weaker notion of optimality, but results in a stronger bound for that weaker notion. We have preliminary results that indicate such results are possible for some join queries (using DLM's techniques). However, it is an open question to provide similar optimality guarantees even for the case of bow-tie queries over a single index.

[5]We would like to thank Kasper Green Larsen and Suresh Venkatasubramanian for answering many questions we had about range search lower bounds and pointing us toward several references.

Acknowledgements

HN is partly supported by NSF grant CCF-1161196. DN is partly supported by a gift from LogicBlox. CR is generously supported by NSF CAREER award under IIS-1054009, ONR awards N000141210041 and N000141310129, and gifts or research awards from American Family Insurance, Google, Greenplum, and Oracle. AR's work on this project is supported by the NSF CAREER Award under CCF-0844796.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] P. K. Agarwal, P. K. Agarwal, J. Erickson, and J. Erickson. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry*, pages 1–56. American Mathematical Society, 1997.

[3] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748, 2008.

[4] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[5] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *Theor. Comput. Sci.*, 239(2):211–229, 2000.

[6] J. Chen, S. Lu, S.-H. Sze, and F. Zhang. Improved algorithms for path, matching, and packing problems. In *SODA*, pages 298–307, 2007.

[7] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, pages 743–752, 2000.

[8] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.

[9] A. Gajentaan and M. H. Overmars. On a class of $o(n^2)$ problems in computational geometry. *Comput. Geom.*, 5:165–185, 1995.

[10] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[11] M. Grohe. The parameterized complexity of database queries. In *PODS*, pages 82–92, 2001.

[12] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM J. Comput.*, 1(1):31–39, 1972.

[13] K. Mehlhorn. *Data Structures and Algorithms*, volume 1. Springer-Verlag, 1984.

[14] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012.

[15] A. Pagh and R. Pagh. Scalable computation of acyclic joins. In *PODS*, pages 225–232, 2006.

[16] C. H. Papadimitriou and M. Yannakakis. On the complexity of database queries. In *PODS*, pages 12–19, 1997.

[17] M. Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proc. 42nd ACM Symposium on Theory of Computing (STOC)*, pages 603–610, 2010.

[18] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.

[19] T. L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[20] D. E. Willard. An algorithm for handling many relational calculus queries efficiently. *J. Comput. Syst. Sci.*, 65(2):295–331, 2002.

[21] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.

APPENDIX

A. ON THE NON-EXISTENCE OF THE BEST DATA-INDEPENDENT TOTAL ORDER OF ATTRIBUTES

This section presents a sample query that shows that there does not exist a data-independent total order of attributes for the algorithm in section 3.3. Consider the query Q_2 from Example 3.1:

$$R(X) \bowtie S_1(X,Y) \bowtie S_2(Y,Z) \bowtie T(Z).$$

Since X and Z play the same role, we will show that for each of the following total attribute orders there exists a database instance for which the order is infinitely worse than another order.

- Bad example for the (Y,X,Z) order. Consider the following instance:

$$\begin{aligned} R(X) &= \{2,\ldots,N\} \\ T(Z) &= \{2,\ldots,N\} \\ S_1(X,Y) &= \{1\} \times [N] \\ S_2(Y,Z) &= [N] \times \{1\} \end{aligned}$$

  The optimal proof for the (Y,X,Z) order needs $\Omega(N)$ inequalities to certify that the output is empty; yet the order (X,Z,Y) needs only $O(1)$ inequalities.

- Bad example for the (X,Y,Z) order. Consider the following instance:

$$\begin{aligned} R(X) &= [N] \\ T(Z) &= [N] \\ S_1(X,Y) &= [N] \times \{1\} \\ S_2(Y,Z) &= \{2\} \times [N] \end{aligned}$$

  The optimal proof for the (X,Y,Z) order needs $\Omega(N)$ inequalities to certify that the output is empty; yet the order (Y,X,Z) needs only $O(1)$ inequalities.

- Bad example for the (X,Z,Y) order. Consider the following instance:

$$\begin{aligned} R(X) &= [N] \\ T(Z) &= [N] \\ S_1(X,Y) &= [N] \times \{1\} \\ S_2(Y,Z) &= \{2\} \times [N] \end{aligned}$$

  The optimal proof for the (X,Y,Z) order needs $\Omega(N)$ inequalities to certify that the output is empty; yet the order (Y,X,Z) needs only $O(1)$ inequalities.

B. MATERIAL FOR BACKGROUND

B.1 BST Background details

B.1.1 Proof of Lemma 2.2

PROOF OF LEMMA 2.2. For notational convenience, define $n \stackrel{\text{def}}{=} |U|$. We first argue that $|W| \leq O(\log n)$. To see this w.l.o.g. assume that $W = [n]$. Thus any node v at level $0 \leq i \leq \log n$, the interval $[\ell_v, r_v]$ is of the form $[j \cdot n/2^i + 1, (j+1)n/2^i]$ for some $0 \leq j < 2^i$. It can be checked that any interval $[\ell, r]$ can be decomposed into the disjoint union of at most one interval per level, which proves the claim.

Next, consider the following algorithm for computing W. We initialize W to be the empty set and call Algorithm 5 with the root of $\mathcal{T}$, ℓ and r.

It is easy to check that Algorithm 5 essentially traverses through the subtree of $\mathcal{T}$ with W as leaves, which by our earlier argument implies that there are $O(\log n)$ recursive calls to Algorithm 5. The claim on the run time of the algorithm follows from noting that each recursive invocation takes $O(1)$ time. □

Algorithm 5 RangeFind(v, ℓ, r)

Input: $v \in \mathcal{T}$ and integers $\ell \leq r$

1: **If** $[\ell_v, r_v] \subseteq [\ell, r]$ **then**
2: Add v to W
3: **Return**
4: Let u and w be the left and right children of v.
5: $[\ell_1, r_1] \leftarrow [\ell, r] \cap [\ell_u, r_u]$
6: **If** $[\ell_1, r_1] \neq \emptyset$ **then**
7: RangeFind(u, ℓ_1, r_1)
8: $[\ell_2, r_2] \leftarrow [\ell, r] \cap [\ell_w, r_w]$
9: **If** $[\ell_2, r_2] \neq \emptyset$ **then**
10: RangeFind(w, ℓ_2, r_2)
11: **Return**

B.1.2 Proof of Lemma 2.5

PROOF. For notational convenience, define $n \stackrel{\text{def}}{=} |U|$, $h \stackrel{\text{def}}{=} |W(I, \mathcal{T})|$ and $m \stackrel{\text{def}}{=} |U_{1\setminus 2}|$.

We begin with the case when $W(I_2, \mathcal{T}) = \emptyset$, then any standard traversal algorithm which starts at the smallest element in $U_{1\setminus 2}$ (which under the assumption is $I_1 \cap U$) and ends at the largest element in $U_{1\setminus 2}$ in time $O(m)$.

Now consider the case when $h > 0$. By Lemma 2.2, we can compute $W \stackrel{\text{def}}{=} W(I_2, \mathcal{T})$ in $O(\log n)$ time. By the fact that we store the minimum and maximum value in $\mathcal{T}_v$ for every vertex v, in $O(h)$ time, for each $u \in W$, we can store the interval $[\ell_u, r_u]$ that we can effectively remove from $U \cap I_1$ that we do not have to worry about. By Remark 2.3, we can assume that these intervals are presented in sorted order.

The rest of the algorithm is to run the usual traversal algorithm while "jumping" over the intervals $[\ell_v, r_v]$ for every $v \in W$. We will assume that the standard traversal algorithm, given a node u in $\mathcal{T}$, one can in $O(1)$ time compute the next vertex in the order. The details are in Algorithm 6.

Algorithm 6 JumpTraverse

Input: BST $\mathcal{T}$, I_1, sorted "jump" intervals $[\ell_v, r_v]$ for $v \in W$

1: Assume the vertices in W by their sorted order are $v_1, \ldots, v_h$.
2: $i \leftarrow 1$.
3: Let u be the left most leaf with value ℓ_{v_1}.
4: Let w be the leaf with the smallest value in $I_1 \cap U$.
5: **While** The value at w is in I_1 **do**
6: $x \leftarrow w$
7: **If** $u = w$ **then**
8: Let x be the rightmost leaf in $\mathcal{T}$ with value r_{v_i}.
9: $i \leftarrow i + 1$
10: Let u be the left most leaf in $\mathcal{T}$ with value ℓ_{v_i}.
11: w is next leaf node after x in the traversal of $\mathcal{T}$.

We now quickly analyze the run time of Algorithm 6. First, note that the loop in Step 5 runs $O(m + h)$ times. Further, the only steps that are not constant time are Steps 3, 4, 8, 10 and 11. However, each of these steps can be done in $O(\log n)$ time using the fact that $\mathcal{T}$ is a BST. This implies that the total run time is $O((m + h) \log n)$, as desired. □

REMARK B.1. *It can be checked that Algorithm 6 (and hence the proof of Lemma 2.5) can be modified suitably to handle the case when given as input disjoint intervals $I_2, \ldots, I_h$ such that $I_j \subseteq I_1$ and we replace $U_{1\setminus 2}$ and $W(I_2, \mathcal{T})$ by $I_1 \setminus \cup_{j=2}^{h} I_j$ and $\cup_{j=2}^{h} W(I_j, \mathcal{T})$. (Note that the unions are disjoint unions.)*

REMARK B.2. *We point out that Algorithm 6 (or its modification in Remark B.1) do not need to know the intervals $I_2, \ldots, I_h$ before the algorithm starts. In particular, as long as the traversal algorithm goes from smallest to largest*

values and we can perform the check (i.e. is w the left most element in the "current" interval) in Step 7 in time at most T, then we do not need to the know the intervals in advance. In particular, by spending an extra factor of T in the run time, we can check in Step 7 if w is indeed the left most element of some interval I_j. If so, we run the algorithm from Lemma 2.2 to compute $W(I_j, \mathbb{F})$ and then we can run the algorithm as before (till we "hit" the next interval $I_{j'}$).

B.1.3 2D BST Background

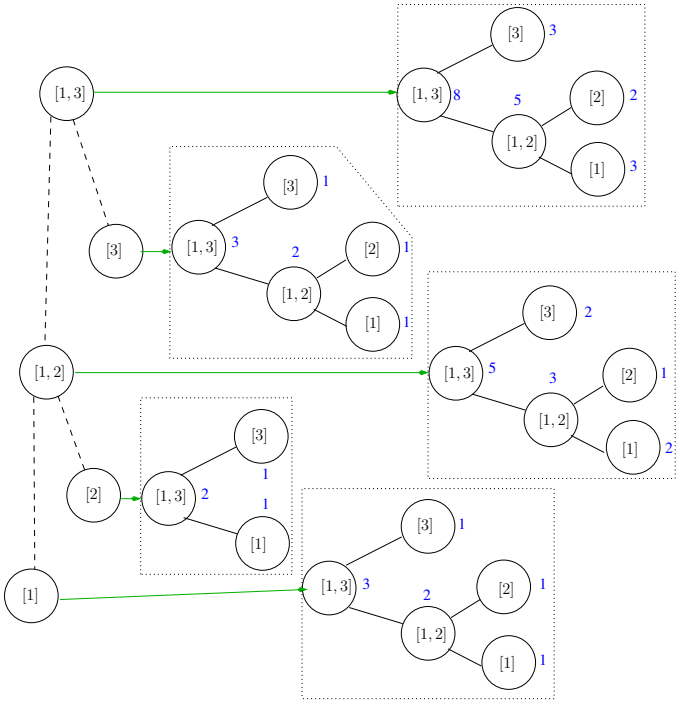

Figure 4: The 2D-BST for the set U in Figure 1. The ranges in each node v represents the interval $[\ell_v, r_v]$. The number in blue near the node its the value n_v. The BST on the x values has dotted lines while the BSTs on he y values have solid lines. Finally for each node v in the x part of the 2D-BST, the BST on the corresponding y values is pointed to by a green arrow from v.

B.2 DLM Background

Definition B.3. *An element e is recursively defined to be eliminated in an argument P if either*

- *$(a < b) \in P$ where e is a weak predecessor of a, and b has no eliminated predecessors;*
- *$(a < b) \in P$ where e is a weak successor of b, and a has no uneliminated successors.*

Lemma B.4. *An argument is a $\emptyset$-proof precisely if an entire set is eliminated.*

Proof. Notes that eliminated elements do not belong to the intersection set. So if an entire set is eliminated, then obviously the argument P is a $\emptyset$-proof.

Now suppose if the argument P is a $\emptyset$-proof, we will show that there is one set which has all elements eliminated. Let's consider the intersection set problem with 2 sets A and B.

Consider the following algorithm that will eliminate entirely one of 2 sets A and B

```
While (A or B is not entirely eliminated ) do
       Denote A[i] and B[j] be the smallest uneliminated elements in A and B.
       If there exists a k >= j such that (B[k] < A[i]) in P then
              eliminate all weak predecessors of B[k] in B
       Else if there exists a k >= i such that (A[k] < B[j]) in P then
```

```
                    eliminate all weak predecessors of A[k] in A.
    End while
```

Note that inside the loop, exact one of 2 conditions must occur because otherwise we can construct an instance satisfying P but its intersection is not empty. So the argument P is not a $\emptyset$-proof.

Also when one of 2 cases is implemented, one of two sets A and B has more elements that are eliminated. Because A and B are finite, the algorithm will stop and it shows that A or B is entirely eliminated.

□

DEFINITION B.5. *A low-to-high ordering of an argument is an ordering with the property that each comparison $(A_s[i] < A_t[j])$ newly eliminates elements just in A_s, unless it entirely eliminates A_s(in which case it may newly eliminate elements in all sets).*

LEMMA B.6. *Every $\emptyset$-proof has a low-to-high ordering.*

PROOF. Consider the $\emptyset$-proof P of the intersection set problem of 2 sets A and B.

We will construct the low-to-high ordering $\emptyset$-proof P' from P as follows:

```
Initialize P' is empty
While (A or B is not entirely eliminated) do
        Denote A[i] and B[j] be the smallest uneliminated elements in A and B.
        If A[i] > B[j] then
              add all comparisons (B[k] < A[i]) in P to P'
        Else add all comparisons (A[k] < B[j]) in P to P'.
End While
Add all remaining comparisons in P to P'.
```

Obviously P' is the low-to-high ordering $\emptyset$-proof.

□

C. ALGORITHMS AND PROOFS FROM SECTION 3.2

C.1 The 1BST Case

In this section, we will consider the following problem: Given 3 relations $R(X)$, $S(X,Y)$, and $T(Y)$, we want to compute $R(X) \bowtie S(X,Y) \bowtie T(Y)$ which is called a bow-tie query. In this section, the relation R and T are sorted by X and Y, respectively. Also the relation S is sorted by X, then by Y. So we call this query bow-tie query in one index case.

C.1.1 Proof structure of bow-tie query in one index case

The main idea to compute the bow-tie query is as follows:

- First we will compute $W(X,Y) = R(X) \bowtie S(X,Y)$. Then for every x, denote $W[x] = \{(x,y)|(x,y)$ is a tuple of $W\}$. Then it is easy to see that W is partitioned into k disjoint relations $W[x_1], W[x_2], \ldots, W[x_k]$.
- The result join of bow-tie query is the union of all $W[x_i] \bowtie T(Y)$.

Notice that we already know how to compute the query $W(X,Y) = R(X) \bowtie S(X,Y)$, which is a hierarchical query that is shown in the Hierarchical Query Section. In the following section, we will focus on how to compute the query $(\bigcup_{i=1}^{k} W[x_i]) \bowtie T(Y)$.

C.1.2 Support for bow-tie query in one index: Union of Intersections Problem

We consider the following problem: Given a set S and a collection of k other sets $A_1, A_2, \ldots, A_k$ that are sorted in the ascending order, we want to compute

$$R = \bigcup_{i=1}^{k} (S \cap A_i).$$

Remark 1: Instead of only output all elements in R, for every element r in R, we also want to output all occurrences of r in every $A_i, i = 1, \ldots, k$.

We call this problem the *union-intersection problem* in this section.

In the following subsections, we will show the algorithm that generates the minimum proof. Then we show that the minimum proof that just involves the comparisons between S and A_i should be optimal within a constant factor. Finally, we describe the algorithm that computes the union-intersection problem and has optimal running time (within a log factor).

C.1.3 Finding the minimum proof

To make the algorithm clean, without loss of generality, we will assume that $S[1] < \min\{A_1[1], A_2[1], \ldots, A_k[1]\}$. Consider the following algorithm that generates the minimum proof for the union-intersection problem.

Algorithm 7 Bow-tie Query Fewest-Comparisons

Input: Instance of union-intersection problem $R = \bigcup_{i=1}^{k}(S \cap A_i)$
Output: A proof P with fewest comparisons, i.e., $|P| = \mathcal{D}$.

1: **While** S is not entirely eliminated and one of $A_i, i = 1, \ldots, k$ is not entirely eliminated **do**
2: Denote $a_{i_1}, a_{i_2}, \ldots, a_{i_m}$ be the minimum uneliminated elements of $A_{i_1}, A_{i_2}, \ldots, A_{i_m}$, respectively.
3: Denote $a = \min\{a_{i_1}, \ldots, a_{i_m}\}$.
4: In the set S, search for the maximum element s such that $s \leq a$.
5: **If** s is the last uneliminated element of S **then**
6: for every $j = 1, \ldots, m$, add the appropriate comparison $(s < a_{i_j})$ or $(s = a_{i_j})$ to the proof P.
7: Stop the algorithm.
8: Let s' be the successor of s in S.
9: Also suppose that $a_{j_1}, \ldots, a_{j_p}$ are all elements in $\{a_{i_1}, \ldots, a_{i_m}\}$ such that $a_{j_t} \leq s'$ for every $t = 1, \ldots, p$.
10: For each $t = 1, \ldots, p$, in the set A_{j_t}, search the maximum element a'_{j_t} such that $a'_{j_t} \leq s'$.
11: Sort $a_{j_1}, a'_{j_1}, \ldots, a_{j_p}, a'_{j_p}$ in the ascending order. Suppose that after sorted, those elements are $b_1, \ldots, b_q$.[6]
12: For notational simplicity, denote $b_0 = s$ and $b_{q+1} = s'$.
13: Add the comparison $(b_0 < b_1)$ to the proof P.
14: Add the comparison $(b_q < b_{q+1})$ to the proof P.
15: **For** $t = 1, \ldots, q-1$ **do**
16: Consider two elements b_t and b_{t+1}, we have the following cases:
17: If there is no i in $[0, t]$ such that b_i and b_{t+1} belongs to the same set and no comparison $(b < b_{t+1})$ is added so far, then add the comparison $(b_t < b_{t+1})$ to the proof P.
18: If there is no i in $[t+1, q+1]$ such that b_t and b_i belongs to the same set, find the smallest index $l, t+1 \leq l \leq q$ such that there is no $j, j < l$ and b_j and b_l belongs to the same set. Also no comparison $(b < b_l)$ is added so far. Add the comparison $(b_t < b_l)$ to the proof P. If no such l is found, add the comparison $(b_t < s')$ to the proof P.
19: Otherwise, nothing is to be added to the proof P.
20: Mark all elements $\leq s$ in S as eliminated.
21: For every $t = 1, \ldots, p$, in the set A_{j_t}, mark all elements $\leq a'_{j_t}$ as eliminated.

THEOREM C.1. *For any given instance $R = \bigcup_{i=1}^{k}(S \cap A_i)$, the algorithm Fewest-Comparisons generates the proof for the union-intersection problem with the fewest comparisons possible.*

PROOF. We show that in each case of the algorithm, it generates the proof with the fewest comparisons possible. Consider the line 5 of the algorithm. If s is the last uneliminated element of S, then for every $j = 1, \ldots, m$, then the fact the a_{i_j} is bigger or equal to s must be specified. Otherwise, P is no longer a valid proof for this instance. It is not difficult to see that m is the most optimal number in terms of number of comparisons needed to add to the proof P to make it valid. So line 5 is efficient.

Consider lines 8-19 in the algorithm. In this case, for every $i = 1, \ldots, q$, the valid proof P must be able to show which of the following facts are true: b_i is bigger to s, b_i is equal to s, b_i is smaller than s', and b_i is equal to s'.

We will prove by induction that at each step in line 17, the algorithm tries to construct the minimum proof by adding those comparisons. Suppose that this fact is true at step $t-1$. Consider the step t.

- Let's look at case a) in line 17. In this case, one of the comparisons $(b_0 < b_{t+1}), \ldots, (b_t < b_{t+1})$ must be added to the proof P. Otherwise the proof P is no longer valid because we can not determine whether $b_{t+1} = s$ or

$b_{t+1} > s$. We can see that if some other minimum proof Q that decides to add $(b_l < b_{t+1})$ to the proof, then we can replace that comparison by $(b_t < b_{t+1})$ and the proof Q is still valid. So in this case, choosing the comparison $(b_t < b_{t+1})$ is optimal.

- Now consider case b) in line 18. We can see that one of the comparisons $(b_t < b_{t+1}), \ldots, (b_t < b_{q+1})$ must be added to the proof P. Otherwise the proof P is no longer valid because we can not determine from P whether $b_t = s'$ or $b_t < s'$. Suppose that l is found and the comparison $(b_t < b_l)$ is added to the proof P. Now suppose that the other minimum proof Q chooses to add the comparison $(b_t < b_h)$. Obviously there exists one comparison $(b < b_l)$ in Q. If $b_h < b_l$ then by replacing $(b_t < b_h)$ by $(b_t < b_l)$, the proof Q is still valid. If $b_h > b_l$ then we can replace two comparisons $(b_t < b_h)$ and $(b < b_l)$ in Q by $(b_t < b_l)$ and $(b < b_h)$ respectively. Note that the proof Q is still a valid minimum proof. So at this case, choosing the comparison $(b_t < b_l)$ is optimal.

  In summary, the algorithm generates the proof with the fewest comparisons.

□

Denote $\mathcal{D}$ be the number of comparisons of the minimum proof that is generated from the algorithm Fewest-Comparisons. Then $\mathcal{D}$ is the lower bound of running time of any algorithm that computes the union-intersection problem. The following theorem shows the upper bound of the size of the minimum proof that contains only the comparisons between S and A_i, and it turns out that this proof is optimal within the constant factor.

THEOREM C.2. *For any instance of union-intersection problem, the number of comparisons in minimum proof that contains only the comparisons between S and $A_i, i = 1, \ldots, k$ is no more than $2\mathcal{D}$.*

PROOF. Let P be a minimum proof and Q be a minimum proof that contains only the comparisons between S and $A_i, i = 1, \ldots, k$. Then we will show that in each phase in the algorithm Fewest-Comparisons, the number of comparisons in Q is no more than two times the number of comparisons in P (1)

Consider line 5 in the algorithm. obviously $|P| = |Q|$, so the fact (1) holds. Consider lines 8-19 of the algorithm, suppose that in $\{i_1, i_2, \ldots, i_m\}$ there are exactly m indices t such that $a_t < a'_t$. So from now in this proof, we call t as an index that has such property and l as an index such that $a_l = a_{l'}$. So we have m such t and $(p - m)$ such that index l.

For every t, 2 comparisons $(s\ \theta\ a_t)$ and $(a'_t\ \theta\ s')$ should be in Q. Also for every l, at most 2 comparisons $(s\ \theta\ a_l)$ and $(a_l\ \theta\ s')$ should be in Q also (when $(a_l = s)$ or $(a_l = s')$ then there is only one of them in Q). (Here $\theta \in \{<, =\}$) So there are at most $2m + 2(p - m) = 2p$ comparisons in the proof Q.

Now we will estimate the lower bound of the number of comparisons in P. For every t, from the proof P, it can be determined whether $a_t > s$ or $a_t = s$. So in P, there are at least one comparison $(b\ \theta\ a_t)$. Because otherwise, it is free to set a_t to be equal or greater than s and also satisfies the proof P; in other words, P is no longer valid. Also for every l, if $a_l < s'$ then in P, there is at least one comparison $(b\ \theta\ a_l)$ so that we can determine whether $a_l > s$ or $a_l = s$ from the proof P. If $a_l = s'$ then there is at least one comparison $(a_l = b)$ so that we can verify $a_l = s'$ from the proof P.

Notice that all comparisons we describe above are pairwise different. So there are at least $m + (p - m) = p$ comparisons. And so, $|Q| \leq 2p \leq 2|P|$.

□

THEOREM C.3. *Given an instance $R = \bigcup_{i=1}^{k}(S \cap A_i)$ of the union-intersection problem, denote $N = max\{|S|, |A_1|, \ldots, |A_k|\}$. Then R can be computed in time $O(\mathcal{D} \log N)$.*

PROOF. Consider the following algorithm Union-Intersection to compute R

Algorithm 8 Union-Intersection

Input: A set S and k sets $A_i, i = 1, \ldots, k$ sorted in the ascending order.
Output: $R = \bigcup_{i=1}^{k}(S \cap A_i)$ (all occurrences of a value in the result need to be output)

1: **For** $i = 1, \ldots, k$ **do**
2: Compute $A'_i = S \cap A_i$ using the Set Intersection Algorithm
3: **return** $R = A_1 \cup \cdots \cup A_k$.

We will show that Algorithm Union-Intersection (Algorithm 8) can compute R in $O(\mathcal{D} \log N)$ time. For every $i = 1, \ldots, k$, denote $\mathcal{D}_i$ be the number of comparisons of the minimum proof of the intersection problem $S \cap A_i$. Then by using Set Intersection Algorithm, we can compute $A'_i = S \cap A_i$ in $O(\mathcal{D}_i \log N)$ time.

Also by Theorem C.2, $\mathcal{D}_1 + \cdots + \mathcal{D}_k \leq 2\mathcal{D}$, so computing all A'_i takes $O(\mathcal{D} \log N)$ time.

By allowing duplications in R, computing R by just outputting all elements in $A_1, \ldots, A_k$ takes $|A_1| + \cdots + |A_k| \leq |\mathcal{D}_1| + \cdots + |\mathcal{D}_k| \leq 2\mathcal{D}$ time.

So R can be computed in $O(\mathcal{D} \log N)$ time. □

C.1.4 Bow-tie query in one index case

Algorithm 9 Bow-Tie-One-Index

Input: Relations $R(X)$ and $T(Y)$ sorted by X and Y respectively. Relation $S(X, Y)$ is sorted by X, then by Y.
Output: $U(X, Y) = R(X) \bowtie S(X, Y) \bowtie T(Y)$

1: $W(X, Y) = R(X) \bowtie S(X, Y)$ by using the hierarchical query algorithm. Partition $W(X, Y)$ into k relations $W_1, \ldots, W_k$ such that for every $i = 1, \ldots, k$, in relation W_i, all tuples have the same X attribute and W_i are sorted by Y in ascending order.
2: $U(X, Y) = \bigcup_{i=1}^{k} (W_i \bowtie T)$ using the algorithm Union-Intersection
3: **return** U

Remark 1: For every $i = 1, \ldots, k$, the algorithm does not have to compute W_i explicitly by adding all satisfied tuples to W_i. Instead, it just has to compute the low index l and high index h such that W_i will contains all tuples $S[l], \ldots, S[h]$. By using the hierarchical query algorithm, it can be obtained and also those tuples are sorted by Y attribute.

Remark 2: Denote $\mathcal{D}_1$ be the number of comparisons of the minimum proof in the hierarchical join query problem $W(X, Y) = R(X) \bowtie S(X, Y)$. Denote $\mathcal{D}_2$ be the number of comparisons of the minimum proof in the Union-Intersection problem $U(X, Y) = \bigcup_{i=1}^{k} (W_i \bowtie T)$. So $\mathcal{D} = \mathcal{D}_1 + \mathcal{D}_2$ is the number of comparisons of the minimum proof in the Bow-Tie query in one index case.

We will show that the Algorithm Bow-Tie-One-Index (Algorithm 9) is optimal within a $\log N$ factor.

THEOREM C.4. *Given an instance $U(X, Y) = R(X) \bowtie S(X, Y) \bowtie T(Y)$ of the bow-tie query in one index case problem; R and T are sorted by X and Y respectively, S is sorted by X, and then by Y. Denote $N = max\{|R|, |S|, |T|\}$. Then U can be computed in time $O(\mathcal{D} \log N)$ time.*

PROOF. By using the hierarchical join query algorithm, we can compute all W_i in $O(\mathcal{D}_1 \log N)$ time.

Also by Remark 1, W_i is sorted by Y. So $T, W_1, \ldots, W_k$ are satisfied to be applied in the algorithm Union-Intersection to compute U. By Theorem C.3, we can compute U in time $O(\mathcal{D}_2 \log N)$.

In summary, U can be computed in time $O(\mathcal{D} \log N)$. □

C.2 An Example where Simple Modification of Algorithm 3 Performs Poorly

Let Q_2 join the following relations:

$$R(X) = [N], \quad S_1(X, X_1) = [N] \times [N], \quad S_2(X_1, X_2) = \{(2, 2)\}, \quad \text{and} \quad T(X_2) = \{1, 3\}$$

Also assume that all relations are sorted in a single index with that order. We run the DFS-Style join algorithm to see how it performs in this example.

The order of attributes are X, X_1, X_2. So for each step, we will search the output tuple by one attribute at a time. First, $R[1] = 1$ and $S_1[1] = [1, 1]$ match by attribute X. But when it considers the relation S_2, it does discover that there is no tuple in S_2 with $X_1 = 1$. So the algorithm backtracks with the next tuple in S_1, that is $S_1[2] = (1, 2)$. Now $S_1[2]$ matches with the tuple $S_2[1] = (2, 2)$. When comparing with X_2 in T, we have the following comparisons: $S_2[1].X_2 > T[1].X_2$ and $S_2[1].X_2 < T[2].X_2$. By those comparisons, the algorithm knows that $S_2[1]$ does not match any X_2 in T. So it backtracks with the next tuple in S_1, that is $S_1[3]$. And it continues with the same fashion. So the above DFS-style algorithm will take $\Omega(N)$ steps to figure out that the result join is empty.

On the other hand, the proof of this instance includes only 2 inequality comparisons $S_2[1].X_2 > T[1].X_2$ and $S_2[1].X_2 < T[2].X_2$. This proof shows that $S_2 \bowtie T(X_2)$ is empty, and hence the Q_2 is empty. But the DFS-style algorithm does not remember those constraints. As a result, it keeps joining among R and S_1, and takes $\Omega(N)$.

D. ALGORITHMS AND PROOFS FOR SECTION 3.3

Before we present our main algorithm in Algorithm 12, we will first present its specialization for the set intersection and the bow-tie query. These specializations are different from Algorithm 1 and 3 respectively. However, unlike the previous algorithms these specialization clearly demarcate the role of the "main" algorithm and the data structure to handle constraints. Further, these specializations will help illustrate the main technical details in our final algorithm.

D.1 Computing the intersection of m sorted sets

The purpose of this section is two-fold. First, we would like to introduce the notion of *certificate* (called *proof* in DLM) that is central in proving instance-optimal run-time of join algorithms. Second, by presenting our algorithm specialized to this case, we are able to introduce the "probing point" idea and provide a glimpse into what the "ruled out regions" and "constraint data structure" are.

Consider the following problem. We want to compute the intersection of m sets $S_1, \cdots, S_m$. Let $n_i = |S_i|$. We assume that the sets are sorted, i.e.

$$S_i[1] < S_i[2] < \cdots < S_i[n_i], \forall i \in [m].$$

The set elements belong to the same domain $\mathbf{D}$, which is a totally ordered domain. Without loss of generality, we will assume that $\mathbf{D} = [N]$. One can think of $[N]$ as actually the index set to another data structure that stores the real domain values. For example, suppose the domain values are strings and there are only 3 strings `this`, `is`, `interesting` in the domain. Then, we can assume that those strings are stored in a 3-element array and $N = 3$ in this case.

In order to formalize what any join algorithm "has to" do, DLM considers the case when the only binary operations that any join algorithm can do are to compare elements from the domain. Each comparison between two elements a, b results in a conclusion: $a < b$, $a > b$, or $a = b$. These binary operations are exclusively used in real-world join implementations. It is possible that one can exploit, say, algebraic relations between domain elements to gain more information about them. We are not aware of any algorithm that makes use of such relationships.

After discovering relationships between members of the input sets, a join algorithm will have to output correctly the intersection. Consequently, any input that satisfies exactly the same collection of comparisons that the join algorithm discovered during its execution will force the algorithm to report the "same" output. Here by the "same" output we do not mean the actual set of domain values; rather, we mean the set of positions in the input that contribute to the output. For example, suppose the algorithm discovered that $S_1[i] = S_2[i] = \cdots = S_m[i]$ for all i, then the output is $\{S_1[1], S_1[2], \cdots\}$, whether or not in terms of domain values they represent strings or doubles or integers. In essence, the notion of "same" output here is a type of isomorphism.

The collection of comparisons that a join algorithm discovers is called an *argument*. The argument is a *certificate* (that the algorithm works correctly) if any input satisfying the certificate must have exactly the same output. More formally, we have the following definitions

DEFINITION D.1. *An* argument *is a finite set of symbolic equalities and inequalities, or* comparisons*, of the following forms: (1) $(S_s[i] < S_t[j])$ or (2) $S_s[i] = S_t[j]$ for $i, j \geq 1$ and $s, t \in [m]$. An instance satisfies an argument if all the comparisons in the argument hold for that instance.*

Arguments that define their output (up to isomorphism) are interesting to us: they are *certificates* for the output.

DEFINITION D.2. *An argument P is called a* certificate *if any collection of input sets $S_1, \ldots, S_m$ satisfying P must have the same output, up to isomorphism. The* size *of a certificate is the number of comparisons in it. The* optimal certificate *for an input instance is the smallest-size certificate that the instance satisfies.*

As in DLM, we use the optimal certificate size to measure the information-theoretic lower bound on the number of comparisons that any algorithm has to discover. Hence, if there was an algorithm that runs in linear time in the optimal certificate size, then that algorithm would be instance-optimal with respect to our notion of certificates.

Our algorithm for set-intersection. Next, we describe our algorithm for this problem which runs in time linear in the size of any certificate. Algorithm 10 is a significant departure from the *fewest-comparison algorithm* in DLM (i.e. Algorithm 1). In fact, our analysis can deal directly with the subtle issue of the role of equalities in the certificate that DLM did not cover. We will highlight this issue in a later example.

The constraint set and constraint data structure. The *constraint set* C is a collection of integer intervals of the form $[\ell, h]$, where $0 \leq l \leq h \leq N + 1$. The intervals are stored in a data structure called the *constraint data structure* such that when two intervals overlap or adjacent, they are automatically merged. Abusing notation, we also call the data structure C. We give each interval one credit, $1/2$ to each end of the interval. When two intervals are merged, say $[\ell_1, h_1]$ is merged with $[\ell_2, h_2]$ to become $[\ell_1, h_2]$, we use $1/2$ credit from h_1 and $1/2$ credit from ℓ_2 to pay for the merge operation. If an interval is contained in another interval, only the larger interval is retained in the data structure. By maintaining the intervals in sorted order, in $O(1)$-time the data structure can either return an integer

Algorithm 10 Computing the intersection of m sets

Input: m sorted sets $S_1, \cdots, S_m$, where $|S_i| = n_i$, $i \in [m]$

1: Initialize the constraint set $C \leftarrow \emptyset$
2: **For** $i = 1, \ldots, m$ **do**
3: Add the constraint $[S_i[n_i] + 1, N + 1]$ to C
4: Add the constraint $[0, S_i[1] - 1]$ to C
5: **While** C can still return a value t **do** ▷ $t \in [N]$
6: **For** $i = 1, \ldots, m$ **do**
7: Let $e_i = S_i[x_i]$ be the smallest value in S_i such that $e_i \geq t$
8: Let $e'_i = S_i[x'_i]$ be the largest value in S_i such that $e'_i \leq t$ ▷ It is possible that $e_i = e'_i$ and $x_i = x'_i$
9: **If** $e_i = t$ for all $i \in [m]$ **then** ▷ Then all $e'_i = t$ too
10: **Output** t
11: Add constraint $[t, t]$ to C
12: **else**
13: **For** each $i \in [m]$ such that $e_i > t$ **do** ▷ $e'_i < t$ and $x'_i = x_i - 1$ for such i
14: Add constraint $[e'_i + 1, e_i - 1]$ to C ▷ This is similar to the 2D-BST case

$t \in [0, N + 1]$ (our *probing point*) that does not belong to any interval, or correctly report that no such t exists. In other words, each query into C takes constant time. Inserting a new interval into C takes $O(\log n)$-amortized time where n is the maximum number of intervals ever inserted into C, using the credit scheme described above.

Certificate. The optimal certificate, or any certificate for that matter, is a set of comparisons of the form $S_i[x_i] \; \theta \; S_j[x_j]$, where $\theta \in \{=, <, >\}$, $i, j \in [m]$, $x_i \in [n_i]$, and $x_j \in [n_j]$, such that any input satisfying the certificate will have the same output, up to isomorphism.

The following theorem states that Algorithm 10 has a running time with optimal data complexity (up to a $\log n$ factor).

THEOREM D.3. *Algorithm 10 runs in time $\tilde{O}(m|P| \log n)$, where P is* **any** *certificate for the instance, and $n = \sum_{i=1}^{m} n_i$.*

PROOF. We first upper bound the number of iterations of the algorithm. The key idea is to "charge" each iteration of the main while loop to a pair of comparisons in the certificate P such that no comparison will ever be charged more than a constant number of times. Each iteration in the loop is represented by a distinct t value. Hence, we will find a pair of comparisons to pay for t instead of paying for the iteration itself.

Let t be a value in an arbitrary iteration of Algorithm 10. Let e_i, e'_i, x_i and x'_i be defined as in lines 7 and 8 of the algorithm.

If $e_i = t$ for all $i \in [m]$, then P has to certify this fact with at least $m - 1$ equalities of the form $S_i[x_i] = S_j[x_j]$. (If we think of the $S_i[x_i]$ as nodes of a graph and the equalities among them as edges, then we must have a connected graph to certify the output t.) It is easy to see that if there were no such equalities then we can slightly perturb the input instance to still satisfy P but t is no longer an output. In this case, we pay for t by charging any of the (at least) $m - 1$ equalities in the certificate.

Now, suppose $e_i > t$ for some i, i.e. t is not an output. (Note that by definition it follows that $e'_i = S_i[x'_i] < t$.) For each $i \in [m]$, the member $S_i[x_i]$ is said to be *t-alignable* if $S_i[x_i]$ is already equal to t or if $S_i[x_i]$ is not part of *any* comparison in the certificate P. Similarly, we define the notion of *t-alignability* for $S_i[x'_i]$, $i \in [m]$.

When $S_i[x_i]$ is t-alignable, setting $S_i[x_i] = t$ will not violate any of the comparisons in the certificate. Similarly, we can transform the input instance to another input instance satisfying P by setting $S_i[x'_i] = t$, provided $S_i[x'_i]$ is t-alignable.

Claim: if t is not an output, then there must exist some $\bar{i} \in [m]$ for which both $S_{\bar{i}}[x_{\bar{i}}]$ and $S_{\bar{i}}[x'_{\bar{i}}]$ are *not* t-alignable. Otherwise, by assigning one t-alignable end in each pair $S_i[x_i]$ and $S_i[x'_i]$ to be t, we obtain a new instance satisfying the certificate P but this time t is an output.

We will pay for t using any comparison involving $S_{\bar{i}}[x_{\bar{i}}]$ and any comparison involving $S_{\bar{i}}[x'_{\bar{i}}]$. Since they are not t-alignable, each of them must be part of some comparison in P. Since we added the interval $[e'_i + 1, e_i - 1]$ to the constraint set C, in later iterations t will never hit the same interval again. Each comparison involving one element will be charged at most 3 times: one from below the element, one from the above the element, and perhaps one

when e_i is output.

We already discussed how the constraint data structure C can be implemented so that insertion takes amortized logarithmic time in the number of intervals inserted, and querying (for a new probing point t) takes constant time. Since the number of intervals inserted is at most $n = \sum_i n_i$, each iteration of the algorithm takes time at most $O(m \log n)$. $\square$

D.2 Instance Optimal Joins with Traditional B-Trees

In this section, we consider the case when every relation is stored as a single B-tree. The B-tree for each relation is assumed to be built consistently with a *global attribute order*. For example, if the global attribute order is $A_1, \cdots, A_n$, and $R(A_2, A_5, A_6)$ is a relation, then the B-tree for relation R has three levels, the first indexed by A_2, the second indexed by A_5, and the third indexed by A_6.

We will start with the first non-trivial query, the *bow-tie query* in order to illustrate some of the core ideas in the algorithm, the constraint data structure, and their analyses. Then, we present the general algorithm and constraint data structure. Finally, if the global attribute order is the reverse elimination order for the input *acyclic* query, then algorithm is instance-optimal in terms of data complexity. Our algorithm can be turned into a worst-case linear time algorithm with the same guarantee as Yannakakis' classic join algorithm for acyclic queries. In Appendix G, we will give an example showing that our algorithm can be much faster than the recent leapfrog triejoin algorithm.

D.2.1 Our algorithm specialized to the bow-tie query

The first non-trivial query is the following which we call the *bow-tie query*:

$$q_{\bowtie} = R(X) \bowtie S(X, Y) \bowtie T(Y).$$

The global attribute order, without loss of generality, can be assumed to be (X, Y). The indices can have different ranges. For example, $R[i]$ is the ith value in R, where i is the index and the value belongs to its own domain. Similarly, $T[j]$ is the jth value in T. As for S, we will use the following notation. $S[i]$ is the ith X-value in S, and $S[i, j]$ is the jth Y-value among all tuples $(x, y) \in S$ with $x = S[i]$. We want to clearly distinguish between the indices into the relations and the values which belong to the domain $[N]$. The reason is that the optimal certificate, or any certificate for that matter, only contains indices and no value in the domain.

Structure of a certificate. Any certificate for the bow-tie query is a set of comparisons in one of the following three formats:

$$\begin{aligned} R[i_r] &\ \theta\ S[i_s], \\ S[i_s, j_s] &\ \theta\ T[j_t], \\ S[i_s, j_s] &\ \theta\ S[i'_s, j'_s]. \end{aligned}$$

where $\theta \in \{<, =, >\}$ is called a *comparison*.

Example D.1. *To see the importance of the third comparison format, consider the following example.*

$$\begin{aligned} R &= [n] \\ S &= [n] \times \{2i \mid i \in [n]\} \\ T &= \{2i - 1 \mid i \in [n]\}. \end{aligned}$$

The following comparisons form a certificate verifying that the output is empty:

$$\begin{aligned} R[i] &= S[i],\ i \in [n] \\ S[i, j] &= S[1, j],\ i, j \in [n], i > 1, \end{aligned}$$

$$T[1] < S[1, 1] < T[2] < S[1, 2] < \cdots < T[n] < S[1, n].$$

The last $2n-1$ inequalities verify that the sets $S[1, *]$ and T do not intersect. There are totally $n+n(n-1)+(2n-1) = n^2 + 2n - 1$ comparisons in the above certificate.

On the other hand, if we do not have comparisons between tuples in S (i.e. comparisons of the kind $S[i_s, j_s]\ \theta\ S[i'_s, j'_s]$), we must at least assert for every i that the sets $S[i, *]$ and T do not intersect, for a total of at least $n(2n-1) = 2n^2 - n$ comparisons.

It is not true that disallowing comparisons between tuples from the same relation will only blow up the certificate by a constant factor. Consider the following example.

EXAMPLE D.2. *Consider the following query, which is not a bow-tie query.*

$$R(A, C) \bowtie S(B, C),$$

where

$$\begin{aligned} R &= [n] \times \{2k \mid k \in [n]\} \\ S &= [n] \times \{2k-1 \mid k \in [n]\} \end{aligned}$$

The join is empty, and there is a certificate of length $O(n^2)$ showing that the output is empty. The certificate consists of the following comparisons:

$$\begin{aligned} R[1, c] &= R[a, c], \text{ for } a, c \in [n], a > 1 \\ S[1, c] &= S[b, c], \text{ for } b, c \in [n], b > 1, \end{aligned}$$

$$S[1,1] < R[1,1] < S[1,2] < R[1,2] < \cdots < S[1,n] < R[1,n].$$

If we don't use any equality, or if we only compare tuples from different relations, any certificate will have to be of size $\Omega(n^3)$ because it will have to "show" for each pair a, b that $R[a, *] \cap S[b, *] = \emptyset$ which takes $2n-1$ inequalities, for a grand total of $n^2(2n-1) = \Omega(n^3)$ comparisons.

Note that our algorithm described below does run in time $\tilde{O}(n^2)$ for this instance.

The constraint data structure. Every constraint is one of the following forms: $\langle [\ell, h], * \rangle$, $\langle = p, [\ell, h] \rangle$, and $\langle *, [\ell, h] \rangle$. A tuple $\mathbf{t} = (x, y)$ satisfies the first form of constraint if $x \in [\ell, h]$, the second form if $x = p$ and $y \in [\ell, h]$, and the third form if $y \in [\ell, h]$. Each constraint can be thought of as an "interval" in the following sense. The first form of constraints consists of all two-dimensional (integer) points whose X-values are between ℓ and h. We think of this region as a 2D-interval (a vertical strip). Similarly, the second form of constraints is a 1D-interval, and the third form of constraints is a 2D-interval (a horizontal strip).

We store these constraints using a two-level tree data structure. In the first level (corresponding to X-values), every branch is marked with an $*$, $= p$, or an interval $[\ell, h]$. In the second level, every branch is marked with $[\ell, h]$. If the first level of a branch is an interval $[\ell, h]$, then there is no second level under that branch. Intervals under the same parent node are merged when they overlap. We use the previous trick of giving the low end and the high end of an interval half a credit to pay for the merging. If the second level of a $(= p)$-branch covers the entire domain, then the $(= p)$-branch is turned into a $\langle [p, p], * \rangle$ constraint that can further be merged (at the first level). We will give half an extra credit to each end point of an interval under a $(= p)$-branch so that when the branch is turned into a $\langle [p, p], * \rangle$ branch both of the end points has half a credit as any other interval on the first level.

Inserting a new constraint takes amortized logarithmic time, as we keep the branches sorted, and the new constraint might "consumes" existing intervals. In amortized logarithmic time, the data structure C has to be able to report a new tuple $\mathbf{t} = (x, y)$ that does not satisfy any of its constraints, or correctly report that no such $\mathbf{t}$ exist. To find $\mathbf{t}$, we apply the following strategy:

- We first find x such that x does not belong to any first level interval and does not match any p in the $(= p)$-branches. This value of x, if it exists, can easily be found in logarithmic time. If x is found, then we find y that does not belong to any second level interval on the $*$-branch. If there is no y then no such $\mathbf{t}$ exist, the algorithm terminates.
- If x from the previous step cannot be found, then the first-level intervals and the values p in the $(= p)$-branches cover the entire domain $\{0, \cdots, N+1\}$. In this case, we will have to set x to be one of the p's, and find a value y under the $(= p)$-branch that does not belong to any interval under that branch. The tuple $\mathbf{t} = (x, y)$ might still violate a $\langle *, [\ell, h] \rangle$ constraint. In that case, we insert the constraint $\langle = p, [\ell, h] \rangle$ into the tree. Then we find the next smallest "good" value y under the $(= p)$-branch again. The intervals under $(= p)$-branch might be merged, but if we give each constraint a constant number of credits, we can pay for all the merging operations.

To summarize, insertion and querying into the above data structure takes at most logarithmic time in the amortized sense (over all operations performed).

The algorithm. Algorithm 11 solves the bow-tie join problem. Notations are redefined here for completeness. Let $|R|, |S|, |T|$ denote the number of tuples in the corresponding relations, $S[*]$ the set of X-values in S, $S[i]$ the i'th X-value, $S[i, *]$ the set of y's for which $(S[i], y) \in S$, and $S[i, j]$ the j'th Y-value in the set $S[i, *]$. Figure 5 illustrates the choices of various parameters in the algorithm.

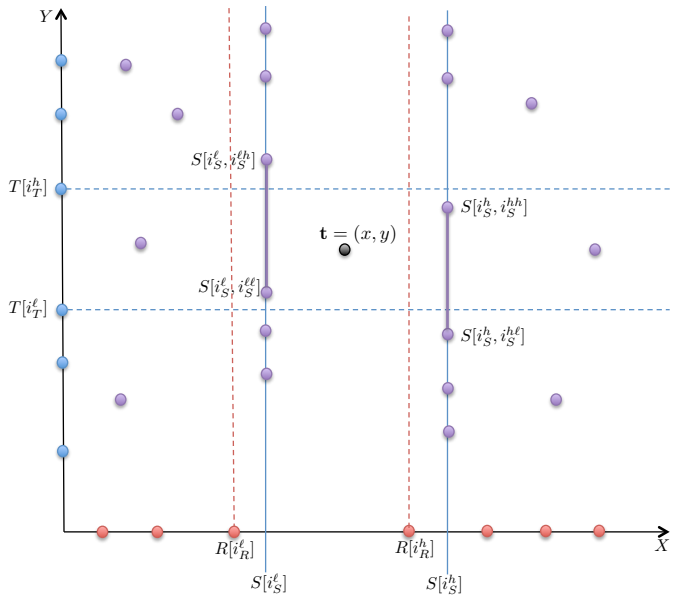

Figure 5: Illustration for Algorithm 11

THEOREM D.4. *Algorithm 11 runs in time $O(|P| \log n)$, where P is* **any** *certificate, and n is the input size.*

PROOF. We pay for each iteration of the algorithm, represented by the tuple $\mathbf{t}$, by charging a pair of comparisons in the certificate P. Each comparison will be charged at most $O(1)$ times. To this end, we define a couple of terms.

Any tuple $e \in \{R[i_R^h], R[i_R^\ell], S[i_S^h], S[i_S^\ell], T[i_T^h], T[i_T^\ell]\}$ is said to be $\mathbf{t}$-*alignable* if either e is already equal to x or e is not involved in any comparison in the certificate P. If a $\mathbf{t}$-alignable tuple e is not already equal to x, setting $e = x$ will transform the input into another input that satisfies all comparisons in P without violating the relative order in the relation that e belongs. A tuple $e \in \{S[i_S^h, i_S^{h\ell}], S[i_S^h, i_S^{hh}]\}$ is $\mathbf{t}$-alignable if $S[i_S^h]$ is $\mathbf{t}$-alignable and either e is already equal to y or e is not part of any comparison in the certificate P. Similarly, we define $\mathbf{t}$-alignability for a tuple $e \in \{S[i_S^\ell, i_S^{\ell\ell}], S[i_S^\ell, i_S^{\ell h}]\}$.

Next, we describe how to pay for the tuple $\mathbf{t}$.

Case 1. Line 9 is executed. Let I be the collection of pairs (i, j) for which we can infer the relation $S[i, j] = S[i_S^h, i_S^{hh}]$ using equalities in the certificate P. In particular, $(i_S^h, i_S^{hh}) \in I$. There must be at least $|I| - 1$ equalities for this inference. There must also be an equality $S[i, j] = T[i_T^h]$ in the certificate for some $(i, j) \in I$, otherwise setting $S[i, j] = y + \epsilon$ for all $(i, j) \in I$ will change the output; in particular, $\mathbf{t}$ will no longer be in the output, yet the new input instance still satisfies P. There are totally $|I|$ equalities involved. We can use one of these equalities to pay for $\mathbf{t}$. If later on $\mathbf{t}' = (S[i], S[i, j])$ is an output for another $(i, j) \in I$, we use a different equality to pay for that output. In total, each equality in the above $|I|$ equalities will be charged at most once for each output of the form $(S[i], S[i, j])$. The constraint added in line 10 ensures that we won't have to pay for the same output $\mathbf{t}$ again.

Case 2. The else part (line 11) is executed. We claim that one of the following five cases must hold:

(1) both $R[i_R^h]$ and $R[i_R^\ell]$ are not $\mathbf{t}$-alignable,
(2) both $S[i_S^h]$ and $S[i_S^\ell]$ are not $\mathbf{t}$-alignable,
(3) both $S[i_S^h, i_S^{h\ell}]$ and $S[i_S^h, i_S^{hh}]$ are not $\mathbf{t}$-alignable,
(4) both $S[i_S^\ell, i_S^{\ell\ell}]$ and $S[i_S^\ell, i_S^{\ell h}]$ are not $\mathbf{t}$-alignable,
(5) both $T[i_T^h]$ and $T[i_T^\ell]$ are not are not $\mathbf{t}$-alignable.

Suppose otherwise that at least one member in each of the five pairs above is $\mathbf{t}$-alignable. For example, suppose the following tuples are $\mathbf{t}$-alignable: $R[i_R^h]$, $S[i_S^\ell]$, $S[i_S^\ell, i_S^{\ell\ell}]$, $T[i_T^h]$. Then, by setting $R[i_R^h] = S[i_S^\ell] = x$ and $S[i_S^\ell, i_S^{\ell\ell}] = T[i_T^h] = y$, we obtain another input instance satisfying all comparisons in the certificate but $\mathbf{t}$ is now an output.

By definition, each tuple e that is not $\mathbf{t}$-alignable must be involved in a comparison in the certificate P. Instead of charging a comparison, we can charge a non $\mathbf{t}$-alignable tuple. If each non $\mathbf{t}$-alignable tuple is charged $O(1)$ times,

Algorithm 11 Evaluating the bow-tie query $R(X) \bowtie S(X,Y) \bowtie T(Y)$.

1: Initialize the constraint set C with the following constraints
- $\langle [R[|R|]+1, N+1], * \rangle$, $\langle [0, R[1]-1], * \rangle$
- $\langle [T[|T|]+1, N+1], * \rangle$, $\langle [0, T[1]-1], * \rangle$
- constraints representing all tuples strictly below the smallest tuple in S, lex order
- constraints representing all tuples strictly above the largest tuple in S, lex order

2: **While** C can still return a tuple $\mathbf{t} = (x, y)$ **do** ▷ $\mathbf{t} \in [N] \times [N]$
3: $\quad i_R^h \leftarrow \min\{i \mid R[i] \geq x\}, \quad i_R^\ell \leftarrow \max\{i \mid R[i] \leq x\}$
4: $\quad i_T^h \leftarrow \min\{i \mid T[i] \geq y\}, \quad i_T^\ell \leftarrow \max\{i \mid T[i] \leq y\}$
5: $\quad i_S^h \leftarrow \min\{i \mid S[i] \geq x\}, \quad i_S^\ell \leftarrow \max\{i \mid S[i] \leq x\}$
6: $\quad i_S^{hh} \leftarrow \min\{i \mid S[i_S^h, i] \geq y\}, \quad i_S^{h\ell} \leftarrow \max\{i \mid S[i_S^h, i] \leq y\}$
7: $\quad i_S^{\ell h} \leftarrow \min\{i \mid S[i_S^\ell, i] \geq y\}, \quad i_S^{\ell\ell} \leftarrow \max\{i \mid S[i_S^\ell, i] \leq y\},$
8: $\quad$ **If** $R[i_R^h] = S[i_S^h] = x$ **and** $S[i_S^h, i_S^{hh}] = T[i_T^h] = y$ **then**
9: $\quad\quad$ **Output** the tuple $\mathbf{t} = (x, y)$
10: $\quad\quad$ Add constraint $(= x, [y, y])$ to C
11: $\quad$ **else** ▷ Interval (a, b) is the same as $[a+1, b-1]$)
12: $\quad\quad C \leftarrow C \cup \left\{ \left\langle (R[i_R^\ell], R[i_R^h]), * \right\rangle, \ \left\langle (S[i_S^\ell], S[i_S^h]), * \right\rangle, \ \left\langle *, (T[i_T^\ell], T[i_T^h]) \right\rangle \right\};$
13: $\quad\quad C \leftarrow C \cup \left\{ \left\langle = S[i_S^h], (S[i_S^h, i_S^{h\ell}], S[i_S^h, i_S^{hh}]) \right\rangle, \ \left\langle = S[i_S^\ell], (S[i_S^\ell, i_S^{\ell\ell}], S[i_S^\ell, i_S^{\ell h}]) \right\rangle \right\}$

then each comparison will be charged $O(1)$-times. We pay for $\mathbf{t}$ by charging any pair of non $\mathbf{t}$-alignable tuples out of the five pairs above. We call each of those five pairs an "interval." The pairs of the type (1), (2), and (5) are 2D-intervals, and the pairs of the type (3), (4) are 1D-intervals.

Due to the constraints added on lines 12 and 13, the 2D-intervals are charged at most once. Since two 2D-intervals might share an end point, each tuple from a 2D-interval might be charged twice. The 1D-intervals are charged at most twice. Each tuple from a 1D-interval might be charged at most four times. □

Thus, for bow-tie queries with a single index we get instance optimal results up to poly log factors.

D.2.2 Algorithm and constraint data structure for general queries

This section generalizes the above ideas to the general case when we want to compute a natural join query of the form

$$q = R_1(\bar{A}_1) \bowtie R_2(\bar{A}_2) \bowtie \cdots \bowtie R_m(\bar{A}_m).$$

To simplify notations, we will try not to use subscripts as much as possible. Let atoms(q) denote the set of relation symbols for query q. (In the above query, atoms(q) = $\{R_1, \cdots, R_m\}$, but we will not use subscripts to indicate relations any more.)

Global attribute order. Let $A_1, A_2, \cdots, A_n$ be sequence of all attributes of q. This sequence is the *global attribute order* which the B-tree index for each input relation has to respect in the following sense. For each relation $R \in$ atoms(q), let arity(R) denote the arity of R. Fix one such relation $R \in$ atoms(q) with arity(R) = k. Let $\bar{A}(R) = (a_1(R), a_2(R), \cdots, a_k(R)) \in [n]^k$ be an integer tuple such that the relation R has k attributes $A_{a_1(R)}, A_{a_2(R)}, \cdots, A_{a_k(R)}$ where $a_1(R) < a_2(R) < \cdots < a_k(R)$. Similar to the previous section, $R[i]$ denotes the ith $A_{a_1(R)}$-value in relation R, sorted in increasing order, $R[i,j]$ denotes the jth $A_{a_2(R)}$-value among all tuples of R whose $A_{a_1(R)}$-value is $R[i]$, and so forth. More generally, let $\mathbf{i} = (i_1, \cdots, i_j)$ be any j-ary positive integer vector with $j \leq k$, then $R[\mathbf{i}]$ denotes the i_j'th $A_{a_j(R)}$-value among all tuples whose $A_{a_s(R)}$-value is $R[i_1, \ldots, i_s]$, for all $s \leq j-1$. Such a tuple $\mathbf{i}$ is called an *index tuple* for relation R.

Structure of certificates. Any certificate for the query q is a set of comparisons of the form

$$R[\mathbf{i}] \ \theta \ S[\mathbf{j}]$$

where $\theta \in \{=, <, >\}$, R and S are two relations in atoms(q) that shares an attribute A_k, and $\mathbf{i}$ and $\mathbf{j}$ are index tuples such that $R[\mathbf{i}]$ and $S[\mathbf{j}]$ are A_k-values. As we have seen from Section D.2.1, it is important to allow the certificates to compare values from the same relation. Hence, in the definition above R and S could be the same relation!

The constraint data structure. The constraints and the constraint data structure becomes much more involved for the general query case considered here. Every constraint is of the following forms: $\langle([N] \cup \{*\})^{i-1} \circ [\ell, r] \circ \{*\}^{n-i}\rangle$, i.e. it has an interval on the attribute A_i (for some $i \in [n]$) with trailing $*$'s and a prefix that has either $*$'s or an element of $[N]$. The entries that are elements of $[N]$ are of the type $(= p)$ as those in Section D.2.1. Appendix D.5 presents the constraints and the data structure in details.

The algorithm. The algorithm is described in the details in Appendix D.3. We present here briefly the main ideas which were alluded to in Section 3.3. Initially, the constraint set C is empty. When asked, C returns a new probing point $\mathbf{t}$ which lies in the output space and which does not satisfy any existing constraints in C. By examining $\mathbf{t}$, each input relation R of arity k inserts into C about 2^k constraints which are meant to rule out a large region containing $\mathbf{t}$ that R itself "knows" for sure there cannot be any output. Overall, there are at most $m2^n$ new constraints added to C per iteration. At least one of these constraints can be "charged" to a pair of comparisons in the optimal certificate in such away that no comparison is charged more than 2^n times. Consequently, the total number of iterations the algorithm loops through is at most $O(2^n|P| + |\text{output}|)$, where P is an optimal certificate. Appendix D presents the algorithm in full and provides a formal proof of the following result.

THEOREM D.5. *The total run time of the algorithm described above is*

$$O\left(mn2^n|P|\log(|input|) + mn\log(|input|)\cdot|output| + T_C\right),$$

where P is **any** *certificate, $|output|$ is the number of output tuples, $|input|$ is the input size, and T_C is the total time it takes the constraint data structure to answer all queries from the algorithm. The algorithm adds a total of $O(m4^n|P| + |output|)$ constraints to C and issues $O(2^n|P| + |output|)$ new probing point accesses to C.*

While it is great that the number of iterations of the algorithm is essentially linear in the size of the optimal certificate, the overall run-time of the algorithm depends on how efficient the constraint data structure is in handling the constraints and answering the queries. With the certificate structure described above, the maximum size of a certificate can only be $O(|\text{input}|^2)$, where $|\text{input}|$ is the input size to the join problem. From there, the following can be shown.

THEOREM D.6. *Unless the exponential time hypothesis is wrong, no constraint data structure can process the constraints and the probing point accesses in time polynomial in the number of constraint inserts and probing point accesses.*

PROOF. We prove this theorem by using the reduction from UNIQUE-CLIQUE to the natural join evaluation problem. The reduction is standard [16]. The UNIQUE-k-CLIQUE input instance ensures that the output size is at most 1. The result of Chen et al. [6] implied that if there was an $O(n^{o(k)})$-time algorithm solving UNIQUE-k-CLIQUE, then the exponential time hypothesis is wrong, and many NP-complete problems have sub-exponential running time. Consequently, if there was a constraint data structure satisfying the stated conditions, the exponential time hypothesis would not hold. □

D.2.3 Instance Optimal Acyclic Queries with Reverse Elimination Order of Attributes

The above negative result requires us to restrict the class of input instances which we hope to design an instance optimal algorithm for. Somewhat expectedly, we are able to do so for the class of *acyclic queries* [1, Ch. 6.4], given that the global attribute order is a reverse elimination order (REO). The REO order will be defined formally in Appendix D.5.2. In this case, the constraint data structure can indeed be implemented efficiently.

THEOREM D.7. *For any acyclic query q with the consistent ordering of attributes being the reverse elimination order (REO), there exists a constraint data structure, denoted by* CONSTRAINTTREE*, that can perform the following operations:*

1. *A constraint can be inserted into the* CONSTRAINTTREE *in amortized time $O(n\log V)$, where V is total number of comparison vectors inserted into T overall.*
2. *This* CONSTRAINTTREE *can return a tuple $\mathbf{t}$ that does not satisfy any constraint in amortized time $O(n2^{3n}\log N)$.*

The implementation of such constraint data structure and its analysis can be found in Appendix D.5. Theorem D.7 and Theorem D.5 prove Theorem 3.7.

D.2.4 Some consequences of our algorithm and comparisons to prior algorithms

Algorithm 12 Evaluate the query $q = \bowtie_{R \in \text{atoms}(q)} R(\bar{A}(R))$

Initialize the constraint set $C = \emptyset$
For each relation $R \in \text{atoms}(q)$ **do**
    Add to C constraints eliminating all tuples above the highest tuple in R
    Add to C constraints eliminating all tuples below the lowest tuple in R
While C can still return a tuple $\mathbf{t} = (t_1, \ldots, t_n)$ **do** ▷ $\mathbf{t} \in [N]^n$
    For each $R \in \text{atoms}(q)$ **do**
        $k \leftarrow \text{arity}(R)$
        For $p = 0$ **to** $k - 1$ **do**
            For each vector $\mathbf{v} \in \{\ell, h\}^p$ **do** ▷ $\{\ell, h\}^0$ contains only the empty vector
                Let $(v_1, \ldots, v_p)$ denote $\mathbf{v}$, then $v_j \in \{\ell, h\}$
                $i_R^{(\mathbf{v},h)} \leftarrow \min\left\{ i \mid R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i\right] \geq t_{a_p(R)} \right\}$
                $i_R^{(\mathbf{v},\ell)} \leftarrow \max\left\{ i \mid R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i\right] \leq t_{a_p(R)} \right\}$
    If $R\left[i_R^{(h)}, i_R^{(h,h)}, \ldots, i_R^{\{h\}^p}\right] = t_{a_p(R)}$ for all $p \in [\text{arity}(R)]$ and for all $R \in \text{atoms}(q)$ **then**
        Output the tuple $\mathbf{t}$
        Add constraint $\langle = t_1, = t_2, \ldots, = t_{n-1}, [t_n, t_n] \rangle$ to C
    else ▷ Interval (a, b) is the same as $[a + 1, b - 1]$)
        For each $R \in \text{atoms}(q)$ **do**
            $k \leftarrow \text{arity}(R)$
            For $p = 0$ **to** $k - 1$ **do**
                For each vector $\mathbf{v} \in \{\ell, h\}^p$ **do**
                    $C \leftarrow C \cup \left\langle = R\left[i_R^{(v_1)}\right], \cdots = R\left[i_R^{(v_1,\ldots,v_p)}\right], \left(R[i_R^{(\mathbf{v},\ell)}], R[i_R^{(\mathbf{v},h)}]\right) \right\rangle$

A worst-case linear-time algorithm for acyclic queries. Yannakakis' classic algorithm for acyclic queries run in time $\tilde{O}(|\text{input}| + |\text{output}|)$. Here, we ignore the small log factors and dependency on the query size. Our algorithm can actually achieve this same asymptotic run-time in the worst-case, when we do not assume that the inputs are indexed before hand. All we have to do is to first index all input relations using REO as the global attribute order, then run our algorithm described above. The algorithm runs in time $\tilde{O}(|P| + |\text{output}|)$ where P is any certificate. It is not hard to show that in the acyclic case there is a certificate P with $|P| = |\text{input}|$. To obtain this certificate, we run Yannakakis' algorithm! Each time a semijoin reducer is run, some tuples are eliminated. The time it takes to eliminate tuples is proportional to the number of eliminated tuples. In order to eliminate a tuple, it must be verified that the tuple cannot be matched with any tuple on the other side of a join. This fact can be verified with two inequalities certifying that the tuple "fall between" two other tuples on the other side of the join. For uneliminated tuples, equalities can be used to certify them. The set of equalities and inequalities constructed this way *is* a certificate because if we run Yannakakis algorithm on any input satisfying these comparisons we would have exactly the same output. This certificate's size is proportional to the time it take to run two rounds of semijoin reducers, which is linear in the input size.

D.3 Main algorithms for general queries, B-tree index case

The algorithm. Algorithm 12 solves the natural join problem with input query q.

THEOREM D.8 (RESTATEMENT OF THEOREM D.5). *The total run time of Algorithm 12 is*

$$O\left(mn2^n|P|\log|input| + mn\log(|input|)\cdot|output| + T_C\right),$$

where P is **any** *certificate, $|output|$ is the number of output tuples, $|input|$ is the input size, and T_C is the total time it takes the constraint data structure to answer all queries from the algorithm. The algorithm adds a total of $O(m4^n|P| + |output|)$ constraints to C and issues $O(2^n|P| + |output|)$ queries to C.*

PROOF. First, let us count the maximum number of iterations by "paying" for the tuples $\mathbf{t}$ which represent each iteration. If $\mathbf{t}$ is an output tuple, then we use a "credit" from the term |output| to pay for this tuple. Hence, the hard part is to account for the tuples $\mathbf{t}$ that is not an output tuple.

Fix a relation $R \in \text{atoms}(q)$ with $\text{arity}(R) = k$ and a number $p \in \{0, 1, \ldots, k-1\}$. For any $\mathbf{v} \in \{\ell, h\}^p$, the value $R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,h)},\right]$ is said to be $\mathbf{t}$-*alignable* if all values

$$R\left[i_R^{(v_1)}\right], \cdots, R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}\right]$$

are already $\mathbf{t}$-alignable **and** if

$$R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,h)}\right]$$

is either equal to $t_{a_{p+1}(R)}$ or it is not involved in any comparison in the certificate P. Similarly, we define $\mathbf{t}$-alignability for the value $R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,\ell)}\right]$. Since $\mathbf{t}$ is not an output tuple, there must be a relation R with arity k, some $p \in \{0, \ldots, k-1\}$ and a vector $\mathbf{v} \in \{\ell, h\}^p$ for which both $R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,\ell)}\right]$ and $R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,h)}\right]$ are **not** $\mathbf{t}$-alignable. For this pair, the constraint added in line 21 is not empty because the pair of values are not equal to $t_{a_{p+1}(R)}$. And, each member of this non-$\mathbf{t}$-alignable pair is involved in a comparison in P. We will pay for $\mathbf{t}$ by charging this pair of comparisons.

Finally, we want to upper bound how many times a pair of comparisons is charged. We think of the pair of comparisons as an interval between

$$R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,\ell)}\right]$$

and

$$R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,h)}\right]$$

in a high dimensional space. To see the charging argument, let us consider a few simple cases. When $p = 0$, then the interval is a "band" from one hyperplane H_1 to another hyperplane H_2. This band consists of all points whose $A_{a_1(R)}$-values are between $R[i_R^\ell]$ and $R[i_R^h]$. We call such an interval an n-dimensional interval. Due to the constraint added in line 21, a non-output tuple $\mathbf{t}$ from a later iteration cannot belong to the band anymore. However, $\mathbf{t}$ might belong to the left of H_1 or the right of H_2, in which case a new n-dimensional interval might be created that is charged to the comparison involving H_1 or involving H_2. Consequently, each comparison from a n-dimensional interval can be charged twice.

When $p = 1$, the interval is an $(n-1)$-dimensional interval which is a band lying inside the hyperplane whose $A_{a_1(R)}$-value is equal to $R[i_R^{(v_1)}]$. In this case, each comparison might be charged 4 times: one from one side of the hyperplane, one from the other side, and twice from the two sides *inside* the hyperplane itself.

It is not hard to formally generalize the above reasoning to show that the comparison involving $R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\right.$ or $R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,h)}\right]$ might be charged 2^{p+1} times. Hence, the total number of iterations is at most $O(2^n|P| + |\text{output}|)$. And, the total number of constraints inserted into the data structure C is likewise at most $O(m4^n|P| + |\text{output}|)$. □

D.4 Running Example

EXAMPLE D.3. *Let Q_2 join the following relations:*

$$R(X) = [N] \qquad S_1(X, X_1) = [N] \times [N]$$
$$S_2(X_1, X_2) = \{(2, 2), (2, 4)\} \qquad T(X_2) = \{1, 3\}$$

In this example, the value domain of every attribute is $[N]$. The algorithm to compute Q_2 will run as follows:

- First the constraint set C is empty.
- For every relation, add constraints that will exclude all tuples below the smallest and above the highest tuple.
  Constraints from S_2 : $\langle *, [1, 1], * \rangle$, $\langle *, = 2, [1, 1] \rangle$, $\langle *, = 2, [5, N] \rangle$, $\langle *, [3, N], * \rangle$
  Constraints from T: $\langle *, *, [4, N] \rangle$
- Step 1: C returns $t = (1, 2, 2)$ which does not belong to any interval.
  Constraints added from T: $\langle *, *, [2, 2] \rangle$
- Step 2: C returns $t = (1, 2, 3)$
  Constraints added from S_2: $\langle *, = 2, [3, 3] \rangle$.

After Step 2, C can not return any tuple. The algorithm terminates.

D.5 The constraints and constraint data structure

D.5.1 Basic Data Structures

We begin with a basic data structure, which we will call SORTEDLIST. Such a data structure L can store N numbers in *sorted* order with the following operations:

1. FIND(v, L) returns (b, i), where $b \in \{0, 1\}$ indicates whether v is in L or not. If $b = 1$, then i denotes the position of v in the sorted order otherwise it denotes the index of the value which is *smaller* than v in the sorted order.
2. GET(i, L) returns the value at the i'th position in the sorted order.
3. INSERT(i, v, L) inserts the value v at the i'th position on the sorted array.
4. DELETE(i, L) deletes the value at the i'th position on the sorted array.
5. DELETEINTERVAL$([\ell, r], L)$ deletes all the values in the interval $[\ell, r]$ contained in the sorted array.

REMARK D.9. *Even though we defined the* SORTEDLIST *data structure for numbers one can of course store more complex elements as long as there is a key value whose domain is totally ordered.*

PROPOSITION D.10. *There exists data structures that implement an* SORTEDLIST L *with* N *elements such that the first four operations above can be performed in time* $O(\log N)$ *in the worst-case and* DELETEINTERVAL *can be implemented in* $O(\log N)$ *amortized time.*

PROOF SKETCH. If we use a BST with any self-balancing structure like Red-Black trees, then the claim on the first four operations follow immediately. For the claim on DELETEINTERVAL note that this implies figuring out the index i' of the smallest $\ell' \geq \ell$ and the index j' of the largest $r' \leq r$ in L. This can be done with a combination of FIND and GET operations and then performing $j' - i' + 11$ many DELETE operations. There might be many of these DELETE operations but since each of them can be amortized against an INSERT operation, we have the claimed $O(\log N)$ amortized runtime for DELETEINTERVAL. □

Next we define the functionality of another data structure that we will use, which we call INTERVALLIST. This data structures stores N intervals $[\ell_i, r_i]$ for $i \in [N]$ (which can be overlapping) and supports the following operations (where I is the INTERVALLIST):

- NEXT(v, I) returns the smallest integer v' such that (i) $v \leq v'$ and (ii) $v' \notin [\ell_i, r_i]$ for every $i \in [N]$.
- CONTAINS$([\ell, r], I)$ returns whether there is an interval that contains $[\ell, r]$.
- INSERT$([\ell, r], I)$ inserts the interval $[\ell, r]$ into I.
- DELETE$([\ell, r], I)$ deletes the interval $[\ell, r]$ from I (we do not insist on $[\ell, r]$ being "covered" any some interval in I).

Next we show that one can use SORTEDLIST to construct an INTERVALLIST with the above operations taking logarithmic *amortized* cost.

PROPOSITION D.11. *There exists an implementation of an* INTERVALLIST *on* N *intervals that implements the* NEXT *and* CONTAINS *operations in* $O(\log N)$ *worst-case time and the* INSERT *and* DELETE *operations in* $O(\log N)$ *amortized cost.*

PROOF. The main idea is to store the N intervals as *disjoint* intervals. The end points are then stored in a SORTEDLIST and then we use the various operations of an SORTEDLIST to implement the operations of INTERVALLIST. Details follow.

At any point of time we maintain $m \leq N$ *disjoint* intervals $[s_i, t_i]$ $(i \in [m])$ such that

$$\cup_{i=1}^{n} [\ell_i, r_i] = \cup_{j=1}^{n} [s_j, t_j].$$

We then store the numbers $s_1 \leq t_1 \leq s_2 \leq t_2 \cdot \leq t_m$ in a SORTEDLIST I (with an extra bit saying whether the number of the left or right end point of an interval). Next we show how we implement the three operations need on I.

We begin with NEXT(v, I). Let $(b, i) \leftarrow$ FIND(v, I) and $u =$ GET(i, I). If u is the right end point of an interval then return $\max(u+1, v)$, else return GET$(i+1, I)+1$. It is easy to check that this implements the NEXT operation correctly and by Proposition D.10, all this can be implemented in $O(\log m) = O(\log N)$ time.

Next, we consider CONTAINS$([\ell, r], I)$. The idea is simple: is we just if both ℓ and r are contained in the same interval. Let $(b_i, i) \leftarrow$ FIND(ℓ, I), $(u_i \leftarrow$ GET(i, I), $(b_j, j) \leftarrow$ FIND(r, I) and $u_j \leftarrow$ GET(j, I). Return true if and

only if $i = j$ and u_i is the left end point of an interval. It is easy to check that Proposition D.10 implies this takes $O(\log N)$ time.

We begin with the DELETE$([\ell, r], I)$ operation. The main idea is to figure out the positions $i' \leq j'$ of the elements in I that belong of intervals that are contained in $[\ell, r]$ and then delete those positrons from I. We now present the details. Let us consider the "left" end first. Let $(b_i, i) \leftarrow$ FIND(ℓ, I) and $u_i \leftarrow$ GET(i, I). If u_i is the left end point of an interval, then we perform INSERT$(i + 1, \ell - 1, I)$ (with $\ell - 1$ as the right end point of the new interval $[u_i, \ell - 1]$) and set $i' \leftarrow i + 2$. If u_i is the right end point of an interval then we set $i' \leftarrow i + 1$. Now we consider the "right" end. Let $(b_j, j) \leftarrow$ FIND(r, I) and $u_j \leftarrow$ GET(j, I). If u_j is the left end point of an interval then we perform INSERT$(j + 1, r + 1, I)$ (with $r + 1$ being the left end point moving from u_j to $r + 1$) and set $j' \leftarrow j$; else we set $j' \leftarrow j$. Finally, we just have to delete all the positions from i' to j' in I. This can be implemented by calling DELETE(i', L) $j' - i' + 1$ many times. To analyze the run time, note that by Proposition D.10, this algorithm above can be implemented in time $O((j' - i' + 1) \log N)$. However, note that each of the $j' - i' + 1$ positions that are deleted must have been added at some earlier point, which leads to an overall $O(\log N)$ amortized runtime.

Finally, we consider the INSERT$([\ell, r], I)$ operation, which works pretty similarly to the DELETE operation. To do this, we first delete all the intervals complete contained inside $[\ell, r]$ and then add the interval (we might have to adjust the end points). We again start from the "left" end. Let $(b_i, i) \leftarrow$ FIND(ℓ, I) and $u_i \leftarrow$ GET(i, I). If u_i is a left end point of an interval then define $\ell' \leftarrow u_i$ else $\ell' \leftarrow \ell$. Now let us consider the "right" end. Let $(b_j, j) \leftarrow$ FIND(r, I) and $u_j \leftarrow$ GET(j, I). If u_j is the left end point of of an interval then define $r' \leftarrow$ GET$(j + 1, I)$ else $r' \leftarrow r$. Then run DELETE$([\ell', r'], I)$. Next, compute $(b, i) \leftarrow$ FIND(ℓ', I) and then run INSERT$(i + 1, \ell', I)$ (as a left end point) followed by INSERT$(i + 2, r', I)$ (as a right end point). By an argument similar to the DELETE case, it can be seen that this algorithm runs in amortized time $O(\log N)$ (and it is easy to see that it correctly implements the INSERT operation). □

D.5.2 Reverse Elimination Order

We begin with the definition of the *reverse elimination order* (henceforth, REO). Given a join query $q = \bowtie_{i=1}^{m} R_i$, we call an attribute A_j to be *private* to R_i if A_j only occurs in R_i. Now consider the following algorithm (which is a minor variation on the well-known GYO algorithm):

- Repeat the following till q is empty:
  1. For every $R \subseteq R'$, remove R. Let the resulting query be q'.
  2. Output all the private attributes (in an arbitrary order) in q'.
  3. Remove all the private attributes in q' and rename it q.

The REO of q is an ordering of the attributes that is the *reverse* of the ordering of the attributes output by the GYO algorithm above.

We will make use of the following simple property of an acyclic query when the attributes in the relations are consistent with an REO.

LEMMA D.12. *Let R and S be two relations in the acyclic query q under REO. If C is a common attribute of R and S, then the following holds. Let A_R (and B_S respectively) be the sequence of attributes (under the REO) of attributes in R (S resp.) till (but not including C). Then either $A_R \subseteq B_S$ or $B_S \subseteq A_R$.*

PROOF. For the sake of contradiction, let us assume that there two distinct attributes A and B such that $A \in A_R \setminus B_S$ and $B \in B_S \setminus A_R$. Note that by the definition of A_R and S_B, both A and B come before C in the REO. However, in the GYO algorithm, it is easy to check that at least one of A or B become private attributes *before* C becomes a private attribute and hence, at least one of A or B has to come after C in the REO, which is a contradiction. □

D.5.3 Constraint Tree

In this section, we present our main data structure, which we call a CONSTRAINTTREE, which we will use to efficiently keep track of all the constraints generated by the various comparisons generated by our probing certificate. Recall that n is the total number of attributes in the join query.

A CONSTRAINTTREE T must support the following two operations:

- INSERT$(u, i, \mathbf{c}, T)$, where v is a "node" in T at "level" i that corresponds to a prefix vector $\mathbf{c}$ of length $n - i$ such that every component is either an interval or in $[N] \cup \{*\}$. that has to be inserted into the subtree of T rooted at v. The comparison $\mathbf{c}$ has a special structure: it has exactly one interval (after which all the positions are $*$– for brevity these trailing $*$'s might be omitted).

- LUB$(\mathbf{x}, T)$ returns the smallest $\mathbf{y} \geq \mathbf{x}$ such that the satisfy all the constraints represented by T.

REMARK D.13. *We note that Algorithm 12 should call* $\mathbf{t} \leftarrow$ LUB$(\mathbf{t}, C)$ *in Step 5. Further, we do not need to satisfy the constraints on* $\mathbf{y}$ *in the definiton above. However if we repeatedly use* LUB *in Algorithm 12, it will produce the output tuples in lexicographic order, which is a good property to have.*

D.5.4 The Data Structure of Constraint Tree

In this section we spell out how we implement a CONSTRAINTTREE.

A CONSTRAINTTREE is a tree with n levels, one for each of the attribute with the root being the first attribute in the global ordering of attributes, the next level corresponds to the next attribute in the ordering and so on. Each node v in T has three data structures:

- CHILD$_v$ is a SORTEDLIST with one entry per child of v in the underlying tree. Each entry in SORTEDLIST is labeled with an element of $[N] \cup \{*\}$ and has a pointer to the subtree rooted at the corresponding child. There are two constraints: if v is a lead then it does not have this SORTEDLIST and second, exactly one label in CHILD$_v$ is $*$. Given a valid label $x \in [N] \cup \{*\}$ we will use CHILD$_v(x)$ to denote the pointer to the corresponding subtree.
- PARENT(v) points to the "parent" of v, which is implied by the CHILD map above.
- INTERVAL$_v$ is a INTERVALLIST of intervals. (These are intervals that have been "ruled" out by comparisons.

We mention two invariant that we always want to maintain about CONSTRAINTTREE:

1. For every node v in a CONSTRAINTTREE, we make sure that none of the labels in CHILD$_v$ is contained in an interval in INTERVAL$_v$.
2. For any path from the root to a node v, the semantics of any interval $[\ell, r] \in$ INTERVAL$_v$ is the following. Let v be at level i and let $\mathbf{c} \in ([N] \cup \{*\})^{i-1}$ be the string of labels in the path. Then any vector $\mathbf{x} \in [N]^n$ that satisfies the following properties: (i) for every $1 \leq j < i$, if $c_j \neq *$, then $x_j = c_j$ and (ii) $x_i \in [\ell, r]$ has been "eliminated" by the set of comparison stored in the tree.

D.5.5 Insertion

We present the algorithm to insert a comparison vector into a CONSTRAINTTREE in Algorithm 13.

Next we argue its correctness and talk about its runtime.

LEMMA D.14. *Algorithm 13 correctly inserts the comparison vector* $\mathbf{c}$ *into the* CONSTRAINTTREE T *in amortized time* $O(n \log N)$*, where* N *is total number of comparison vectors inserted into* T *overall.*

PROOF. Recall that we are only dealing with comparison vectors $\mathbf{c}$ which are of the form $([N] \cup \{*\})^{i-1} \circ [\ell, r] \circ \{*\}^{n-i}$, i.e. it has an interval for the ith attribute (for some $i \in [n]$) with trailing $*$'s and a prefix that has either $*$'s or an element of $[N]$. Algorithm 13 uses the prefix to go down the corresponding path in T and then insert the interval at the appropriate INTERVAL. Note that by the invariant mentioned in Section D.5.4 such an algorithm is correct. Thus, to prove the correctness of Algorithm 13, we argue that the algorithm does indeed follow the outline above. Steps 3 and 11 allow the algorithm to go down the corresponding path in T for the prefix of size $i-1$ and Step 13 inserts the interval $[\ell, r]$ into the appropriate location. (Step 14 then removes all children of v that have labels in $[\ell, r]$ since they have now been eliminated.) The rest of the algorithm takes care of the remaining "border" cases. First we note that if a value $c_j \neq *$ is in INTERVAL$_w$ for some node w at level $j \leq i-1$, then it means the current comparison is subsumed by existing constraints and Steps 5 and 6 do this check. Further, if at node w there is no child corresponding to the value c_i (assuming of course that c_i is not contained in INTERVAL$_w$, which we have already checked for), then we need to create the child node. This is achieved by Step 10. Finally, we have to take care of the case that the current set of constraints in T rule out all possible tuples: this is achieved by Step 15 and subsequent steps.

Propositions D.10 and D.11 imply that every step (except Steps 3 and 11) in Algorithm 13 can be implemented in (amortized) time $O(\log N)$.[7] In other words, we perform $O(\log N)$ amortized work in each recursive call. Further, it is each to check that Algorithm 13 traces a path in T and each recursive call makes the algorithm go to the next level. This implies that there are at most n recursive calls, which with the above argument implies the claimed runtime. □

[7]We note that each comparison vector adds either one entry to the CHILD SORTEDLIST or one interval to the INTERVAL INTERVALLIST to any node in T. Since there are N comparison vectors in total, each CHILD$_v$ and INTERVAL$_v$ for any node v has at most $O(N)$ elements, which by Propositions D.10 and D.11 implies the claimed $O(\log N)$ time.

Algorithm 13 InsertTree$(v, \mathbf{c}, i, T)$

Input: A ConstraintTree T, a node v in T (at level i) and a comparison vector $\mathbf{c} = (c_1, \ldots, c_n)$.

Output: Update the data structure with $\mathbf{c}$. Output FULL if the subtree rooted at v has eliminated all possible tuples

$u \leftarrow$ NULL ▷ Flag used to check for a FULL
If c_i is $*$ **then**
  $u \leftarrow$ InsertTree(child$_v(*), \mathbf{c}, i+1, T)$
If c_i is u_i **then**
  If contains$([u_i, u_i],$ interval$_v)$ **then**
    Return NULL ▷ $\mathbf{c}$ is subsumed by an exiting constraint
  else
    $(b, w) \leftarrow$ Find$(u_i,$ child$_v)$ ▷ Checking is u_i is a valid child label
    If $\neg b$ **then**
      Insert$(w+1, u_i,$ child$_v)$ ▷ Create a new node for child with label u_i
    $u \leftarrow$ InsertTree(child$_v(u_i), \mathbf{c}, i+1, T)$
If c_i is $[\ell, r]$ **then**
  Insert$([\ell, r],$ interval$_v)$ ▷ Insert the interval into interval list
  DeleteInterval$([\ell, r],$ child$_v)$ ▷ Update the child$_v$
If u is FULL **then** ▷ All tuples in the child subtree have been exhausted
  If c_i is $*$ **then**
    Return FULL ▷ This node has also exhausted all possibilities.
  else If c_i is u_i **then**
    Insert$([u_i, u_i],$ interval$_v)$.
If contains$([0, N],$ interval$_v)$ **then**
  Return FULL
else
  Return NULL

D.5.6 Lub : New Definitions

We start with some definitions. We think of [=] branches at the left and [$*$] branches at the right.

- A node s at level i in the tree can be identified by a tuple of length i where each component of the tuple is one of two types: (1) in the format $[= c_i]$ where $c_i \in [N]$, called an *equality*, or [$*$], called a *star*. We call this tuple the *path* to s and denote the tuple by $P(s)$.
- A *specialization* of a node s is a node v in the tree such that the path to v can be obtained from the path to s by substituting [$*$] values that occur in $P(s)$ with equalities. We require that there is at least one such substitution (so that u is not a specialization of itself). Said another way, v's path ($P(v)$) satisfies (1) if $P(v) = [*]$ then $P(s)_i = [*]$ and (2) if $P(s)_i$ is $[= c]$ then $P(v)_i$ is $[= c]$.
- A node s is a *specialization wrt* $\mathbf{x}$ of v if for every extra equality in s (as compared to v), say at attribute A_i is of the form $= x_i$.
- An *interval* I in the tree at height $n - h$ can be described by a tuple $(v, [l, u])$ where v is a node at height $n - h$ and $l \leq u$ are in $[N]$.
- We say that an element $\mathbf{x} \in [N]^n$ intersects an interval $(v, [l, u])$ at level i if $(x_1, \ldots, x_{i-1})$ matches $P(v)$ and $x_i \in [l, u]$.
- Given intervals $I = (s, [l, u])$ and $J = (v, [l', u'])$, we say that J is to the right of I if whenever $P(s)_i = *$ then $P(v)_i = *$ and $P(v)$ contains strictly more stars than $P(s)$.
- It follows immediately that if v is a specialization of s then s is strictly to the right of v.

D.5.7 Lub : The Algorithm

We present our algorithm to compute the LUB of a given frontier vector $\mathbf{x}$ w.r.t. a ConstraintTree T in Algorithm 14.

Algorithm 14 LUB$(\mathbf{x}, u, i, T)$

Input: A ConstraintTree T, vector $\mathbf{x} \in [N]^n$ and u is a node in T at level $i \in [n]$.
Output: A lub of $\mathbf{x}$ w.r.t. T.

1: $x' \leftarrow$ NEXT$(x_i, \text{INTERVAL}_u)$.
2: **If** $x' > x_i$ **then** ▷ x_i lies in an interval
3: $y_j \leftarrow x_j$ for $j = 1, \ldots, i-1$
4: $y_i \leftarrow x'$ ▷ Advance the frontier at i from x_i to x'
5: $y_k \leftarrow 0$ for $k = i+1, \ldots, n$.
6: **For** v a specialization of u wrt $\mathbf{x}$ **do**
7: InsertFull$(v, [x_i, x' - 1], i, T)$ ▷ We identify u as the killer of $[x_i, x'_i]$ in v
8: $\mathbf{x} \leftarrow (y_1, \ldots, y_n)$
9: **While** TRUE **do**
10: $\mathbf{y} \leftarrow$ LUB$(\mathbf{x}, \text{CHILD}_u(*), i+1, T)$.
11: $\mathbf{y}' \leftarrow$ LUB$(\mathbf{y}, \text{CHILD}_u(y_i), i+1, T)$. ▷ We have ensured $\mathbf{x} \le \mathbf{y} \le \mathbf{y}'$.
12: **If** $\mathbf{x} = \mathbf{y}'$ **then**
13: **Return** $\mathbf{x}$
14: $\mathbf{x} \leftarrow \mathbf{y}'$ ▷ Some branch advanced (conflict). Check again.

Algorithm 15 InsertFull$(v, [l, u], i, T)$

Input: A ConstraintTree T, v is a node in T and an interval $[l, u]$.
Output: Insert $[l, u]$ and handle if its full

1: InsertTree$(v, [l, u] \circ \{*\}^{n-i}, i, T)$ ▷ For whatever portion is inserted, that part is considered dead (and murdered as above)
2: **If** CONTAINS$([0, N], \text{INTERVAL}_v)$ **then**
3: HandleFull(v, T)

Algorithm 16 HandleFull(v, T)

Input: A v is a node in a tree T.
Output: A Tree Updated with v
1: **If** v is the root **then**
2: **Return** ▷ Done! The algorithm is finished.
3: $p \leftarrow$ PARENT(v).
4: **If** v is a $[*]$ branch of p **then** ▷ p is now full too!
5: HandleFull(p, T)
6: **If** v is reached via the $[= c]$ branch of p **then**
7: InsertFull$(p, [c, c], i-1, T)$.

We will argue the following:

Theorem D.15. *For acyclic queries with the consistent ordering of attributes being the* REO*, Algorithm 14 correctly computes the* LUB *of* $\mathbf{x}$ *w.r.t.* T *in amortized time* $O(n2^{3n} \log N)$.[8]

Proof. We defer the runtime analysis to Section D.5.8.

The argument for the correctness is fairly straightforward. We present a sketch for the sake of completeness. If x_i falls in INTERVAL_v, we must advance $\mathbf{x}$ at level i to beyond the right endpoint of the corresponding interval. This is exactly what Steps 2-5 do. Step 7 just broadcasts the knowledge of the interval just discovered to the relevant nodes in T. (The importance of this step comes to the fore in the runtime analysis in Section D.5.8).

If x_i does not fall into any interval then we need to explore both the corresponding $[*]$ branch and the $[=]$ branch. This is exactly what Steps 9-14 do. In particular, in Step 10, we explore the $[*]$ branch and in Step 11 we explore

[8]Here the amortization is over *both* all the calls to LUB over the entire run of Algorithm 12 as well as all the intervals inserted by Algorithm 12 because of comparisons.

the [=] branch. (Again this ordering will be useful in our runtime analysis.) If we make no progress in either branch, we return in Step 13 else we continue the explorations by updating the frontier in Step 14.

We are done except we need to prove the correctness of Algorithm 15 and 16. The argument for correctness of Algorithm 16 is exactly the same argument we used to handle FULLs in proof of correctness of Algorithm 13 and is omitted. Given the correctness of Algorithm 16 and 13, the correctness of Algorithm 15 follows immediately.

To complete the proof, we note that whenever the algorithm advances the frontier, it does so by the minimum possible amount and hence, it computes the LUB correctly. □

D.5.8 Runtime Analysis of Algorithm 14

We begin with a set of crucial definitions.

DEFINITION D.16. *An* INSERTFULL *operation is* triggered *by an interval in the* NEXT *call. An interval I is* dead*, if it is the result of an* INSERTFULL *operation, and I does not overlap with any live interval in that corresponding interval list.*

More precisely, let us consider the case when $[\ell, r]$ is added to a node v due to an INSERTFULL *at node v. We consider two cases:*

- *$[\ell, r]$ intersects with $[\ell_1, r_1]$ in INTERVAL_v on the left and this is the only intersection with any interval in INTERVAL_v. $[r_1 + 1, r]$ is the new dead interval (which is killed by the interval that triggered the* INSERTFULL*). If $[\ell_1, r_1]$ was originally live, then $[\ell_1, r_1]$ remains live.*
- *$[\ell, r]$ intersects with live intervals $[\ell_1, r_1]$ on the left and $[\ell_2, r_2]$ on the right. Then we replace the three intervals by $[\ell_1, r_2]$, which is now considered live.*
- *All original intervals in the* CONSTRAINTTREE *T are live.*
- *One can convert any live interval into a* cashed-in *interval. (We will come back to this when we prove Proposition D.22.)*

We make the following observation about Algorithm 12. Every interval that is inserted into T has the following property. If the comparison vector inserts the interval $[l, r]$ at node v for attribute C, then there exists a relation R such that the C is an attribute of R. Further, the prefix of the attribute set of R before C exactly matches the sequence of attributes corresponding to the equalities in $P(v)$. For notational convenience, define this prefix to be $\text{PREF}(v, C)$. Further, let the relation R be denoted by $\text{REL}(v)$.

Now note that by the above description of which intervals are live and dead, we can only have live intervals at nodes v where Algorithm 12 inserts a interval. We finally record a property that will be crucial later:

LEMMA D.17. *If there exists two distinct nodes u and v in T such that both have a live interval for the same attribute then either u is a specialization of v or v is a specialization of u.*

PROOF. We first note that since both $u \neq v$ have a live interval for the same attribute (let's call it C), then $\text{REL}(u) \neq \text{REL}(v)$. Further, we note that u is a specialization of v iff $\text{PREF}(v, C) \subset \text{PREF}(u, C)$. Finally, Lemma D.12 completes the proof. □

Killing Mappings. We construct a map $\Phi(I, z) = J$, called the *killing map*, that is implicitly constructed in Algorithm 14. The killing map takes an interval I and a scalar value z contained in that interval i.e. $I = (s, [l, u])$ and $z \in [l, u]$ to another interval $J = (v, [l', u'])$ such that $z \in [l', u']$–and v is strictly to the right of s.

We say that J killed (I, z). We want to keep track at a fine granularity of who killed each part of an interval (ping-pong-ing may create a larger interval that is to blame). Informally, this mapping is described in the comments of Algorithm 14.

PROPOSITION D.18. *Killing mappings exist and are well-defined.*

PROOF. For every dead interval, we claim killing mappings exist.

By inspection of the algorithm, the frontier is only advanced for two reasons:

1. an interval in the current list moves it forward as of the result to a call to NEXT in Step 1. In this case, there is a single interval responsible for this move, say J and we say that J is responsible for all the values in the frontier for $x_i \in J$. In this case, all intervals inserted by in Step 7 of Algorithm 14 are killed by J.
2. as a result of a list filling up, in which case HANDLEFULL takes care of this. We will handle FULLs separately later in our argument.

□

DEFINITION D.19. *We say J **recursively killed** (I, z) if either $\Phi(I, z) = J$ or $\Phi(I, z) = J'$ where J recursively killed (J', z).*

PROPOSITION D.20. *Suppose we encounter an* LUB *computation a node v and $\mathbf{y} < \mathbf{y}'$ (as described in Algorithm 14). Then, for every element in $q \in [\mathbf{y}, \mathbf{y}']$, there must exist an interval J under an equal branch of v that is live and intersects q.*

PROOF. Suppose not, let $I = (s, [l, u])$ at level h be the smallest interval that advances the LUB on the [=] branch (the $\mathbf{y}'$ probe). Such an interval must exist since the LUB is advanced by the [=] branch (and $\mathbf{y} < \mathbf{y}'$). Moreover, there must be some element $z \in [l, u]$ that is larger than $\mathbf{y}$, i.e. substituting the stars.[9] Let z be the smallest such element. Such an interval and z must exist (otherwise the interval I would contain all the information and no advancement of the LUB would be needed). If this interval is alive, we are done. So we may assume without loss of generality that I is dead.

Since the killing map is defined, every J that recursively killed (I, z) and is under the [=] branch of v must also be dead (otherwise we are done, as we've found the required live interval). However, eventually (after at most n steps) the action continually moves right in the tree until we arrive at a J that is under the [$*$] branch of v and killed some interval under the [=] branch. However, this implies a contradiction: we show that the lub on [$*$] branch should have moved past $\mathbf{y}$ (where it was claimed to have ended): in particular let $q = \mathbf{y}$ except that $q_h = z$. Then $q \geq \mathbf{y}$ but q is ruled out by the interval J, so the LUB on the [$*$] branch should have moved past it, a contradiction. In fact, given the above and the fact that the interval J that killed I is always a superset of the latter, we can do the argument till $z = u$ above. In other words, if the frontier advances from $\mathbf{y}$ to $\mathbf{y}^1$ due to I above, we have take care of all $q \in [\mathbf{y}, \mathbf{y}^1)$.

Now consider the the next time the frontier in the [=] branch advances from $\mathbf{y}^1$ to $\mathbf{y}^2$. Note that in this case, this advancement has to happen to the *left* of s. Thus, we can do the same argument as above to match every $q \in [\mathbf{y}^1, \mathbf{y}^2)$ to a live interval in the [=] branch of some descendant of s (and hence trivially under the [=] branch of s). We can inductively continue this argument to finish the proof of the claim. □

REMARK D.21. *It might be useful for the later proofs to remember that when $\mathbf{y}'$ advances in the [=] branch, it might advance the different attributes at different calls to* LUB. *Proposition D.20 implies that for every advancement of $\mathbf{y}'$, no matter for which attribute, there is a live interval in the [=] branch at the same level. In particular, for the current $\mathbf{y}'$ and every attribute i, there is a corresponding live interval by Proposition D.20 at level i (in T) under the [=] branch of some v.*

Informally, the first pass through the while loop to find a live interval is free (nothing has to get killed). However, for our charging argument to work, if we repeat the while loop at a particular level and advance, then we need to argue that *some* live interval will be cashed-in. In turn, we use this to bound the total number of such conflicts.

We use the notation $\mathbf{y}(v, t)$ ($\mathbf{y}'(v, t)$ resp.) to indicate the value of $\mathbf{y}$ ($\mathbf{y}'$ resp.) after at a node v after t iterations of the while loop. We introduce the notion of *cashing in* to analyze the algorithm. A live interval may be cashed-in. We will maintain that a live interval $[l, u]$ will immediately precede a dead interval, i.e., the smaller endpoint of the dead interval will be $u + 1$. Intervals will be cashed-in with each conflict.

PROPOSITION D.22. *Suppose there is a conflict at iteration $t + 1$ at node v in its while loop that is $\mathbf{y}'(v, t) < \mathbf{y}'(v, t+1)$–then at least one live interval is cashed-in the [=] branch of v during the lub computation for $\mathbf{y}'$. Moreover, no interval is cashed-in twice.*

PROOF. Let $\mathbf{y}' = \mathbf{y}'(v, t)$, that is the final state of $\mathbf{y}'$ before we "cross" over to the [$*$] branch of v in the $(t+1)$th iteration of the while loop. Now, since there is a conflict, $\mathbf{y}$ advances under the [$*$] branch. Let us assume that the *first* advancement happens for attribute i. Thus, Proposition D.20 (and Remark D.21) implies that there exists a live interval J that corresponds to the last advancement of $\mathbf{y}'$ in the LUB algorithm at level i. (Let w denote this node where $\mathbf{y}'$ advances.) Now, consider the *first* time $\mathbf{y}$ advances in the [$*$] branch. (Recall that this happens at level i.)

Let this happen at node u. We first claim that the interval that advances $\mathbf{y}$ at u has to be live. The argument is similar to that used in Proposition D.20: If $\mathbf{y}$ advances due to a dead interval then we move right from u using the killing map. Note that we can keep on doing this till we either hit a live interval (and this is the interval that should

[9] Define $q_i = d_i$ if c_i is $= d_i$ and $q_i = y_i$ if $d_i = *$ for $i = 1, \ldots, h-1$, $q_h = z$ and $q_j = 0$ for $j = h+1, \ldots, n$.

have advanced $\mathbf{y}$ already). If we do not hit only dead intervals then ultimately, we will at a node a such that $P(a)$ only has $*$s. Since a is not a specialization of any node, it is easy to check a can only have live intervals. Thus, we can assume that u has a live interval.

Now since u and w both have live intervals, Lemma D.17 implies that w has to be a specialization of u.

Due to Step 7 in Algorithm 14 will cause several InsertFulls– in particular, in all nodes at specializations at level i to the left of u in the $[*]$ branch. In particular, this inserts a dead interval that is an *immediate successor* of J in w, since w is a specialization of u. We will cache in J below (since J is both live and is immediately followed by a dead interval). This shows that a valid cashed-in interval exists to be charged.

To show that an interval is cashed-in once. We assume, for contradiction, that some interval $I = (s, [l, u])$ (at the ith level) is cashed in-twice. However, this implies that the interval I (after being cashed-in) advances the lub for some $\mathbf{y}'$, and since it will be cashed-in again, $y'_i = u + 1$. However, since the cashed-in interval is contiguous with a dead interval that starts at $u+1$, the y'_i would advance to at least $u+2$ the "second time" around. This contradicts that I would be cashed-in again. Thus, any live interval is cashed-in at most once. □

In turn, this implies that the number of conflicting runs through any given while loop at a particular height is bound by the number of original live intervals, which is N. Thus, the overall number of conflicting operations is bound by nN.

Moreover, we observe that this implies we will run through each loop at most twice when there is no conflict (perhaps we will advance on the $[=]$ branch and then have no conflict with the $[*]$ branch which is two lub calculations).

Actual Running Time. We start the run time analysis by assuming that there are no conflicts *and* there are no fulls generated.

Conflict-Free and full*-free Case.* We first observe that if we the lub advances without conflicts this implies that in every while loop at every node v either we advance from the $[*]$ loop or the $[=]$ node but not both. Using the above that we will invoke at most 4 lub calls at each level, our running time is $4^n = 2^{2n}$ invocations. Each such invocation could spawn at most 2^n insert statements. Thus our running time is $O(2^{3n} \log N)$. We stress that we did not try to optimize this bound.

Next, we remove the assumption of no conflicts.

Conflict and full*-free Case.* If there is a conflict, then by Proposition D.22, there is a unique live interval that we cash-in. Since the number of unique live intervals is upper bounded by the original number of intervals, number of conflicts is bound by nN as argued above. This yields a running time of

$$O(2^{3n} \log N(1 + nN)) = O(n2^{3n} N \log N),$$

since the time between any two conflict is bounded by the runtime bound in the case above.

Finally, we consider the general case.

Conflict and full *Case.* We start with the two main observations that complete the runtime analysis for the general case:

1. By looking at the statement of InsertTree algorithm, note that whenever a HandleFull is called at node v, we delete the subtree of T rooted at an ancestor u of v (where $u = v$ is allowed) and we insert a new interval in parent(u).
2. Proposition D.20 implies the following: for any node v, if there is a conflict in the subtree rooted at v, then the live interval to which we charge this conflict also lies in the same sub-tree.

The above implies that we can do the run time analysis as follows: we divide the run of the algorithm into *phases*, for between two fulls. Note each phase can have conflicts but cannot have a full, i.e. it falls in the previous case. Further, (and more importantly), intervals that are in a subtree that is dropped never gets used again in the run of the algorithm. Thus, in some sense after one full, in the next phase it is as if the algorithm was always working with the new ConstraintTree. The catch of course is that we need to somehow "pay" for the new interval that is inserted by HandleFull. However, note that modulo this payment we can just run the analysis above over *all* phases and get the same run time.

Finally, we discuss how we pay for the new intervals inserted by HandleFull. Note that we can only insert a new interval via HandleFull if there is at least one original interval in the subtree that is thrown away in between two phases. In other words, the total number of fulls that we can have at any level is at most N. Thus, over all

levels, we might end up with nN FULLs. (This is because we could potentially "charge" the same original interval for n FULLs– one for each level that HANDLEFULL algorithm "climbs" in T.) Now, since each such interval might be charged one conflict, we will need to add overall to the run time the same bound as we did in the previous case, which implies that we still have the same asymptotic run time of $O(n2^{3n}N \log N)$.

E. MISSING PROOFS FROM SECTION 4.1

E.1 Proof of Theorem 4.2

PROOF OF LEMMA 4.3. Given (x, y), first run binary search on R. If $x \notin R$, then we know the largest number $r_i < x$. Then set $\ell_x = r_i$ and $r_x = r_{i+1}$ to obtain the corresponding X-hole. Otherwise $x \in R$ and we run binary search on T with y. If $y \notin T$, then we can compute the corresponding Y-hole as above. Otherwise we know case (i) is true. Since we only run constant many binary searches, the time bound follows. □

We now finish the proof of Theorem 4.2. Next, we again consider another special case: in particular, let us assume that $h_Y = 0$ (and we know this). However, unlike the previous case we do not know about the X-holes in advance. The idea is to run the algorithm from the previous special case but build the set of X-holes on the fly. By making sure that the traversal algorithm for a 1D-BST iterates through the leaves from the smallest to the largest value, we can assume (that in the absence of any X-holes), the algorithm iterates over all elements in S in order of the X values (for for each fixed $x \in X$, it iterates through all the (x, y) in order of the y values). Further, by Lemma 4.3 in $O(\log n)$ time, we can check whether an (x, y) is in an X-hole or not (and if so then compute the interval corresponding to the X-hole). Thus, by Remark B.2 and the argument for the earlier special case, we can conclude that we can solve the case of $h_Y = 0$ in time $O((h_X + |\mathcal{O}|) \log m \log n)$.

Next, we consider our final special case. Let us assume that we know that $h_Y > 0$ and the Y-gaps are known up front– $I_2, \ldots, I_{h_Y+1}$. Let $y_{\min}$ and $y_{\max}$ denote the smallest and largest y values in T. (Note that we can compute these values in time $O(\log n)$.) Let $L_1, \ldots, L_{h_Y+1}$ be the *live intervals*, i.e. these are the intervals in the set $[y_{\min}, y_{\max}] \setminus \cup_{j=2}^{h_Y+1} I_j$. Note that the final output is

$$\mathcal{O} = \cup_{j=1}^{h_Y+1} (R \times T) \cap (S \cap L_j).$$

For notational convenience define $S_j \stackrel{\text{def}}{=} S \cap L_j$ and $\mathcal{O}_j = (R \times T) \cap (S \cap L_j)$. Then note that $\mathcal{O}$ is the disjoint union of $\cup_{j=1}^{h_Y+1} \mathcal{O}_j$. Further, $\mathcal{O}_j$ has *no* Y-holes. In other words, if we had access to a 2D-BST for every S_j, then using the algorithm for the special case above, we could have computed $\mathcal{O}_j$ in time $O((h_X + |\mathcal{O}_j|) \log m \log n)$, which means we can solve the entire problem in time $O((h_X h_Y + |\mathcal{O}|) \log m \log n)$.

Thus, to finish the proof we need to show (i) how we can simulate a 2D-BST for S_j from the given 2D-BST for S and (ii) remove the assumption that we know the Y-holes (and hence the live intervals) beforehand. We begin with (i). We first note that given a vertex v in $\mathcal{T}^X$, by Lemma 2.6, we can in $O(\log m)$ time compute a succinct description of which y values in $\mathcal{T}^Y(v)$ still lie in S_j (and in particular, we know whether this number is non-zero or not). Thus, we can simulate a traversal on the 2D-BST for S_j by traversing the 2D-BST for S as before but not going down the subtree of any vertex v for which we know that there are no values in S whose y coordinate lies in L_j. Note that this would traverse only a constant factor more nodes than when we had a perfect 2D-BST for S_j and we would spend $O(\log m)$ factor more time (due to Lemma 2.6) in each node of $\mathcal{T}^X$. Thus, in the worst case, applying this trick on the algorithm in the paragraph above will lead to an overall time of $O(((h_X + 1)(h_Y + 1) + |\mathcal{O}|) \log^2 m \log n)$, as desired.

We finally argue how we get rid of the assumption of knowing the Y-holes beforehand. The trick is the same one that we used earlier for X-holes. We will compute the set of live intervals in an online fashion. In particular, we will have a set of candidate live intervals $\mathcal{L}$. We initialize $\mathcal{L} = [y_{\min}, y_{\max}]$. We run the algorithm till $\mathcal{L} = \emptyset$ and at any point of time we work with the "left-most" candidate live interval L in $\mathcal{L}$. If at any point, we encounter a new new Y-hole during our traversal, we split L into (at most two) new live intervals L_1 and L_2 (where L_1 is the left most among the two). We then continue the algorithm with L_1 and put L_2 back into $\mathcal{L}$. The only thing we have to careful about is not to repeat whatever part of the traversal we have done so far L for L_2. To avoid this, when we add L_2 to $\mathcal{L}$, we retain the *state* of the traversal so that when we work with L_2 at a later point, we can just re-start the traversal for L_2 at the point when L got "split." To finish we make the simple observation we only need to store $O(\log m)$ amounts of information to maintain the state.

LEMMA E.1. *It is enough to maintain a path of length $O(\log m)$ in the 2D-BST for S as the current state of traversal in the 2D-BST. It takes $O(1)$ time to compute the next node in the traversal (and to update the state).*

E.2 Proof Sketch of Theorem 4.4

The proof of Theorem 4.4 is pretty much the same as that of Theorem 4.2 so we just sketch where the proof differs. Note that every "live" interval L considered in Algorithm 4 belongs to $\mathcal{L}_Y$. Now note that Algorithm 4 basically goes through all points in S that lie in L but jumps over all the X-holes it encounters. (There are two remarks. First, Algorithm 4 does not know about all the X-holes in $L \sqcap \mathcal{X}$ up front. However, the algorithm "discovers" such an X-hole when it encounters the first element in the hole and hence, we can w.l.o.g. assume that the algorithm knows all the X-holes in $L \sqcap \mathcal{X}$ up front. Second, Algorithm 4 at some point might work with a live intervals that cover more than one interval in $\mathcal{L}_Y$. However, this only happens when the algorithm has not discovered the Y-hole(s) that fall between these multiple intervals in $\mathcal{L}_Y$. Hence, we can again assume w.l.o.g. that the algorithm knows the intervals in $\mathcal{L}_Y$ up front.) Thus, if $\mathcal{O}_\ell \subseteq \mathcal{O}$ are the output points that fall in $\ell \in \mathcal{L}_Y$, then the algorithm spends time (up to poly-log factor) $|\ell \sqcap \mathcal{X}| + |\mathcal{O}_\ell|$ dealing with points in ℓ. The discussion above then implies the claimed runtime in Theorem 4.4.

E.3 Comparison with Previous Algorithms: Details

Proof of Proposition 4.12. Given that $h_X = h_Y = 2$ and that the output is empty, Theorem 4.2 implies that Algorithm 4 will takes time $O(\log^3 n)$ on the instances in Example 4.1. On the other hand, let us consider our Algorithm 3 where assume that S is first indexed by X and then Y (a similar argument holds when we first sort by Y and then X). The first step in this algorithm will be to compute $R \cap \pi_X(S)$, which note is same as R. In other words, just this first step will take time $\Omega(n)$, which is exponentially worse than the runtime of Algorithm 4. Finally, we consider the NPRR algorithm. In this case the best fractional cover of the query hyper graph is to pick S to the extent of 1 and to pick R and S to the extent of 0. In other words, we go through each tuple $(x, y) \in S$ and check if $x \in R$ and $y \in T$. Note that this will also take $\Omega(N)$ time. □

Proof of Proposition 4.13. We begin with NPRR. We first note that NPRR on the bow-tie query runs in time $O(\min(|R| \cdot |T|, |S|))$. Since $h_X \leq |R|$ and $h_Y \leq |T|$, Theorem 4.2 and Corollary 4.5 implies that Algorithm 4 never runs slower then NPRR (up to poly-log factors).

We now consider Algorithm 3. For notational convenience, we assume that the one index for S is first sorted by Y and then by X. (If not, one can suitably restate the run time of Algorithm 4 to make the rest of the argument go through.) Then Algorithm 3 first computes $\mathcal{Y}' = T \cap \pi_Y(S)$. Note that by definition, $|\mathcal{Y}'| \geq |\mathcal{L}_Y|$. Further, for each $y \in \mathcal{Y}'$, we perform the set intersection $R \cap S(X, y)$. We group these intersection as $\cup_{y \in L \cap \mathcal{Y}'} R \cap S(X, y)$ for every $L \in \mathcal{L}_Y$. We claim that the total time taken on these intersections is $\Omega(|L \sqcap \mathcal{X}| + |\mathcal{O}_L|)$. This claim along with Theorem 4.4 then implies that Algorithm 4 is no worse (up to poly-log) factors than Algorithm 3, as desired. Finally, we argue the claim. The main observation is that the DLM algorithm on $R \cap S(X, y)$ takes time (up to logarithmic factor) at least linear in the number of X-holes that cover some element in $S(X, y)$ (as well as the number of pairs $(x', y) \in \mathcal{O}$). This is because every such X-hole exactly corresponds to a jump/gap in R in the DLM algorithm. Summing up over all $y \in L \cap \mathcal{Y}'$ implies the claim (note that the total number of relevant X-holes encountered is at least $|L \sqcap \mathcal{X}|$). Thus, we have argued that □

E.4 Missing Lower Bound Proofs

Proof of Lemma 4.8. To explain the main idea, let us first assume that algorithm **A** can only issue *non-adaptive* box queries, i.e. **A** cannot design the next box query based on the answers to the previous queries.

We construct an instance of the problem that can "fool" **A** if it issues two few queries. Let $n = |\mathcal{X}|$, $m = |\mathcal{Y}|$, and $s = |\mathcal{S}|$. We first define $\mathcal{X}$ and $\mathcal{Y}$, and specify $\mathcal{S}$ later:

$$\begin{aligned} \mathcal{X} &= \{(2i - 3/2, 2i - 1/2) \mid i \in [n]\} \\ \mathcal{Y} &= \{(2j - 3/2, 2j - 1/2) \mid j \in [m]\} \end{aligned}$$

In other words, $\mathcal{X}$ is the collection of n open intervals of length 1 centered at the odd integers $2i - 1$, $i \in [n]$; and $\mathcal{Y}$ is the collection of m open intervals of length 1 centered at the odd integers $2j - 1$, $j \in [m]$. Points in S, which we have yet to specify, will have non-negative integral coordinates. Hence, for the purposes of bounding the number of queries **A** makes, without loss of generality we can assume that every box query that **A** issues has the form $\{[x_1, x_2], [y_1, y_2]\}$ where $x_1, x_2, y_1, y_2 \in \mathbb{Z}$. (If a box query does not satisfy the property then we can simply replace it by a "rounded" query satisfying the property with identical answers.)

A 2×2 *gadget* is a set of four locations where we will put a couple of points in S. In particular, for every pair of indices i, j where $i \in [n]$ and $j \in [m]$, define the 2×2 gadget to be the set of four locations:

$$G_{i,j} = \{(2i - 1, 2j - 1), (2i - 1, 2j), (2i, 2j - 1), (2i, 2j)\}.$$

Note that of the four points in $G_{i,j}$, only $(2i, 2j)$ is not covered by $\mathcal{X}$ and $\mathcal{Y}$, and the other three are covered. Now, define

$$\begin{aligned} G^0_{i,j} &= \{(2i-1, 2j), (2i, 2j-1)\} \\ G^1_{i,j} &= \{(2i-1, 2j-1), (2i, 2j)\}. \end{aligned}$$

And, suppose $S \cap G_{i,j}$ is either $G^0_{i,j}$ or $G^1_{i,j}$. Then, if a box query contains an even number of points from the 2×2 gadget $G_{i,j}$, then the box query won't be able to distinguish whether $S \cap G_{i,j}$ is $G^0_{i,j}$ or $G^1_{i,j}$.

Now, let q be the total number of box queries that $\mathbf{A}$ issues to the data structure. Without loss of generality, assume $4q < mn$ and $8q + 2 \leq |S|$; otherwise we are done. For any box query Q, the number of 2×2 gadgets that Q has an odd number of points in is at most 4 (at the 4 corners of the query). We choose points in S so that, for each 2×2 gadget G of those (at most) four gadgets at the corners of Q, $S \cap G = G^0$. Thus far we have used up at most $8q$ points in S. Since $4q < mn$ there is still at least one gadget $\bar{G}$ that is not at any corner of the q box queries. We use the next two points in S to put in $\bar{G}$ so that $S \cap \bar{G}$ is either $\bar{G}^0$ or $\bar{G}^1$. The rest of the points in S can be put at points covered by $\mathcal{X}$ and $\mathcal{Y}$ that are far away from the gadget. Whether $S \cap \bar{G}$ is $\bar{G}^0$ or $\bar{G}^1$, the results of the box queries are identical and hence $\mathcal{A}$ cannot tell whether S is covered.

Finally, when $\mathbf{A}$ can issue adaptive query, our strategy is the same: we put points in S at arbitrary q gadgets G such that $S \cap G = G^0$. In the end, there must be at least one gadget $\bar{G}$ *not* covered by the q adaptive queries. At that point we can switch so that $S \cap \bar{G} = \bar{G}^1$. □

Proof of Lemma 4.9. The returned point set of a box query to $\mathcal{D}$ is a collection of subtrees of the original dyadic tree $\mathcal{D}$. Each subtree is itself a dyadic 2D-BSTs. Now, each time $\mathbf{A}$ visits a node of a subtree, we can unambiguously view the visit as another box query corresponding to points in that sub-tree. Hence, the situation is exactly the same as in Lemma 4.8, except that we charge each node visit to be a box query. Hence, the bound in the number of box queries in Lemma 4.8 becomes a run-time bound for $\mathbf{A}$ in this case. □

Proof of Lemma 4.10. For convenience, for each $i \in [d]$, define $n_i = |\mathcal{X}_i|$, and

$$\mathcal{X}_i = \{(2j_i - 3/2, 2j_i - 1/2) \mid j_i \in [n_i]\}$$

For each tuple $\mathbf{j} = (j_1, \cdots, j_d) \in [n_1] \times \cdots \times [n_d]$, define a *d-cube gadget*

$$G_{\mathbf{j}} = \{(x_1, \cdots, x_d) \mid x_i \in \{2j_i - 1, 2j_i\}, \forall i \in [d]\}.$$

Note that there is exactly one point in $G_{\mathbf{j}}$ that is not covered by the $(\mathcal{X}_i)_{i=1}^d$; that is the point $(2j_1, 2j_2, \cdots, 2j_d)$.

Finally, define $G^0_{\mathbf{j}}$ to be the set of points $(x_1, \cdots, x_d) \in G_{\mathbf{j}}$ such that the number of indices $i \in [d]$ for which $x_i = 2j_i$ has different parity from d. And, define $G^1_{\mathbf{j}}$ to be the set of points $(x_1, \cdots, x_d) \in G_{\mathbf{j}}$ such that the number of indices $i \in [d]$ for which $x_i = 2j_i$ has the same parity as that of d. Formally,

$$G^0_{\mathbf{j}} = \{(x_1, \cdots, x_d) \in G_{\mathbf{j}} \text{ such that } |\{i \in [d] : x_i = 2j_i\}| - d = 1 \pmod 2\},$$

and

$$G^1_{\mathbf{j}} = \{(x_1, \cdots, x_d) \in G_{\mathbf{j}} \text{ such that } |\{i \in [d] : x_i = 2j_i\}| - d = 0 \pmod 2\}.$$

Note that the sole point $(2j_1, \cdots, 2j_d)$ that $(\mathcal{X}_i)_{i=1}^d$ covers belongs to $G^1_{\mathbf{j}}$.

Now, similar to the proof of Lemma 4.8, suppose for some $\mathbf{j}$ we have $S \cap G_{\mathbf{j}}$ is either $G^0_{\mathbf{j}}$ or $G^1_{\mathbf{j}}$. We claim the following is true: if a d-dimensional box query $Q = (I_1, \ldots, I_d)$ does not contain exactly one point from the gadget $G_{\mathbf{j}}$, then from the answer to Q we cannot tell whether $S \cap G_{\mathbf{j}}$ is $G^0_{\mathbf{j}}$ or $G^1_{\mathbf{j}}$.

To see this, suppose Q contains more than 2 points from $G_{\mathbf{j}}$. This means there has to be at least one index $i \in [d]$ for which both $2j_i - 1$ and $2j_i$ belong to I_i. Now, let K be the (non-empty) set of all indices $i \in [d]$ for which both $2j_i - 1$ and $2j_i$ belong to I_i. This means $[d] - K$ is the set of indices i for which either $2j_i - 1$ or $2j_i$ belong to I_i, but not both. From these observations, it is easy to see that $|Q \cap G^1_{\mathbf{j}}| = |Q \cap G^0_{\mathbf{j}}| = 2^{|K|-1}$, proving the claim.

For each box query Q, there are at most 2^d cube gadgets G that has exactly one point in Q. We will make sure that $S \cap G = G^0$ for those corner gadget. The rest of the proof proceeds as that of Lemma 4.8. □

E.5 Proof of Lemma 4.7

We will need the following lemma from [17]. The statement of the lemma is slightly different from Lemma 11 in [17] but this one follows immediately from the proof of Lemma 11 in [17].

LEMMA E.2. *Assume 3SUM required $\Omega(n^2/f(n))$ expected time and let R be such that*

$$\omega(f(n \cdot f(n))\sqrt{n}) < R < o(n/(f^2(n \cdot f(n)))).$$

Then $\Omega(n^2/f^2(n \cdot f(n)))$ expected time is needed to report $O(n^2/R)$ triangles in the following tripartite graph (for any x that divides n):

- *the three parts are A, B, C of sizes $|A| = Rx$, $|B| = Rn/x$ and $|C| = n$*
- *each vertex in C has at most x neighbors in A and at most n/x neighbors in B*
- *There are $O(nR)$ edges between A and B.*

We note that Lemma 11 in [17] is stated for $x = \sqrt{n}$.[10] For the sake of completeness we sketch the part of the proof of Lemma E.2 that differs from the proof of Lemma 11 in [17].

PROOF PROOF SKETCH. [17] shows that there exists a near linear hash function $h : U \to [R]$ such that each bucket has $O(n/R)$ elements mapped to it. Further, for any $r \in [R]$, let $h^{-1}(r)$ denote the pre-image of r under h. (Note that $|h^{-1}(r)| \leq O(n/R)$.) The hardness result in [17] follows by solving the following problem:

> Given $2nR$ triples (x, y, i), which represent the edges between A and B, check whether $h^{-1}(y)$ intersects $h^{-1}(x + y) - i$ or $h^{-1}(x + y + 1) - i$.

We now present the construction of the tripartite graph. First let $A = [R] \times [x], B = [R] \times [n/x]$ and $C = [n]$. For any $(a, i) \in A$ interpret it as the set $h^{-1}(a) - i$ and interpret any $(b, j) \in B$ as the set $h^{-1}(b) + j \cdot x$. There is an edge from (a, i) to c iff $c \in h^{-1}(a) - i$ and there is an edge from (b, j) to $c \in C$ iff $c \in h^{-1}(b) + j \cdot x$. As mentioned above there are $2nR$ checks to be performed, each of the form: does $h^{-1}(y)$ intersect $h^{-1}(x + y) - s$? First, this intersection is encoded as an intersection between $h^{-1}(y) + \lfloor s/x \rfloor \cdot x$ and $h^{-1}(x + y) - s \mod x$. The latter is then encoded as an edge between $(y, \lfloor s/x \rfloor) \in B$ and $(x + y, s \mod x) \in A$.

The claims on sizes of A, B and C as well as the number of edges between A and B follows from the construction above. Next, we verify the claim on the degree of elements in C. It is easy to check that the construction above implies that each $c \in C$ has an edge to $(h(c + i), i) \in A$ for every $0 \leq i \leq \min(n - c, x)$ (which implies that c has at most x neighbors in A) and an edge to $(h(c - j), j) \in B$ for every $0 \leq j \leq \min(c, n/x)$ (which implies that c has at most n/x neighbors in B).

The rest of the proof is exactly the same in the proof of Lemma 11 in [17]. □

Next, we present the simple implication of solving a bow-tie query to listing triangles in a tripartite graph.

LEMMA E.3. *Let G be a tripartite graph on tripartition (A, B, C) such that there are at most m edges between A and B, at most t triangles in G and every $c \in C$ has at most $\deg_A(c)$ neighbors in A and $\deg_B(c)$ neighbors in B.*

Also assume that any bow-tie query such that all of the relations R, S and T have sizes at most N can be solved in time $O((g(h_X, h_Y) + |\mathcal{O}|) \cdot \log^d N)$ for some constant $d \geq 1$.

Then all triangles in G can be listed in time

$$O\left(\left(\sum_{c \in C} g(\deg_A(c), \deg_B(c)) + t + m + \sum_{c \in C}(\deg_A(c) + \deg_B(c))\right) \log^d \max(m, |A|, |B|)\right). \tag{2}$$

PROOF. Let S denote the set of edges between A and B. (Note that $|S| = m$.) For every $c \in C$ let R_c (and T_c) denote the set of neighbors of c in A (B resp.). For every $c \in C$, define the bow-tie query:

$$\mathcal{O}_c = R_c \bowtie S \bowtie T_c.$$

It is easy to check that the set of triangles in G is exactly the output of the query

$$\cup_{c \in C} \mathcal{O}_c.$$

[10] There is another technical difference: Lemma 11 in [17] assumes the hardness of the related problem of Convolution-3SUM hardness of $\Omega(n^2/f(n))$. However, Theorem 10 in [17] shows that an $\Omega(n^2/f(n))$ hardness of 3SUM implies a $\Omega(n^2/f^2(n \cdot f(n)))$ hardness for Convolution-3SUM hardness, which in turn explains the $f^2(n \cdot f(n))$ terms in Lemma E.2.

To solve the above we solve the bow-tie query $\mathcal{O}_c$ as follows. We first build a 2D-BST for S in time $O(m \log m)$ and then sort the elements in R_c (and T_c) in time $O(\deg_A(c) \log(\deg_A(c))) \leq O(\deg_A(c) \log(|A|))$ ($O(\deg_B(c) \log(|B|))$ resp.). Solving the bow-tie queries for every $c \in C$ by assumption takes time

$$O\left(g(\deg_A(c), \deg_B(c)) + |\mathcal{O}_c|\right) \log^d N,$$

where define $N = \max(m, |A|, |B|)$. Thus, summing up all these times along with the facts that $t = \sum_{c \in C} |\mathcal{O}_c|$ and that we only need to build the 2D-BST for S only once implies the claimed running time in (2). □

We now collect all the results above to prove Lemma 4.7.

Proof of Lemma 4.7. We apply Lemma E.3 to the hard instance from Lemma E.2. Note that in this case we have for every $c \in C$, $h_X = \deg_A(c) \leq x$ and $h_Y = \deg_B(c) \leq n/x$). Thus, we have that $g(h_X, h_Y) \leq n^{1-\epsilon}$. Further, we have $t = O(n^2/R)$ and $m = O(nR)$. Thus, if we pick R and $\max(x, n/x) < o(n/(\text{poly}(\log n) f^2(n \cdot f(n))))$, then Lemmas E.3 and E.2 implies that the 3SUM problem can be solved in time $O(n^{2-\epsilon/2})$, which would contradict Conjecture 4.6, as desired. □

F. ILLUSTRATION FOR ALGORITHM 4

See Figure 6 for a snapshot of the run of Algorithm 4 on the bow-tie query from Figure 3.

G. (NON)INSTANCE OPTIMALITY OF LEAPFROG-JOIN

In this section, we will summarize how Leapfrog join algorithm from [19] works and how it is related to our adaptive join algorithm. Leapfrog join has two parts. The first part is about the algorithm to compute the intersection of k sorted sets. The second part is about using the intersection algorithm to compute the general join query.

First we will summarize their intersection algorithm. The basic idea is similar to the DLM adaptive intersection set. Specifically, as in DLM, the element is eliminated in low-to-high ordering fashion. At any point of time, the set of smallest uneliminated elements in all sets is maintained. Suppose that a_1 is the smallest and a_2 is the largest element in that set; also $a_1 \in A_1$ and $a_2 \in A_2$. Then in the set A_1, the algorithm will gallop to find the smallest element a_1' such that $a_1' \geq a_2$. It turns out that all elements between a_1 and a_1' (not including a_1') are eliminated. And the smallest uneliminated element in A_1 is now set to a_1' and the algorithm continues by the same fashion. In summary, this algorithm is nothing different from DLM algorithm; the only difference is that the average case is analyzed in DLM by introducing the notion of gap to obtain the better running time (by galloping in both sides instead of just from low to high)

Second part the Leapfrog join algorithm to compute the general join query is introduced. The algorithm works as follows. First we need to have a global ordering of all attributes. Then for every relation, it will be sorted in a single index in which their attributes' order is consistent to the global order. To make it simple, we will describe how it works with triangle query $R(A, B) \bowtie S(B, C) \bowtie T(A, C)$. Here the global order is chosen to be A, B, and C and so R is sorted by A, then by B. Similar case is applied for both S and T. To compute this triangle query, first they use the intersection set algorithm in the first part to compute $\pi_A(R) \cap \pi_A(T)$. Then for every a in that set, the intersection set algorithm is used again to compute $\sigma_{A=a}(R) \cap \pi_B(S)$. Then again, for every b in that set, we compute $\sigma_{B=b}(S) \cap \sigma_{A=a}(C)$. Finally we have the output tuple (a, b, c) for every satisfied element a,b, and c.

It is claimed that the above algorithm is optimal in the worst case, but within the log factor. It is fairly easy to see that the above algorithm is not instance optimal for acyclic queries. Consider Example 3.1. Recall that in the example, the join is $R(X) \bowtie S(X, Y) \bowtie S_2(Y, Z) \bowtie T(Z)$, where $|S_2| = |T| = 1$ and $S_2 \bowtie T = \emptyset$. Then our algorithm can discover that the result join will be empty within the constant time (up to log factor) by running the elimination process between S_2 and T. But the Leapfrog join algorithm may take longer to discover that the result join is empty because their steps are as follows: first the join over attribute X is computed, then for every x, compute y, and finally for every y, compute z. Hence it would take $\Omega(N)$ time, which is not instance optimal.

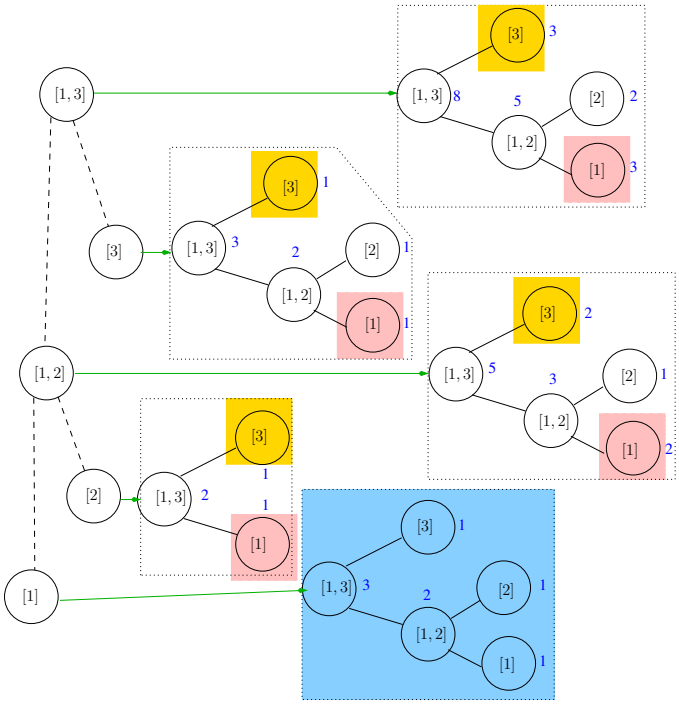

Figure 6: Run of Algorithm 4 on the bow-tie query from Figure 3. We are at a stage when the X-hole corresponding to $x = 1$ and the Y-hole corresponding to $y = 1$ have been discovered. The discovered X-hole is denoted by the light blue box while he Y-hole is denoted by the pink box. (Ignore the yellow boxes for now.) At this point the live interval is $[2, 3]$. The algorithm will first check the BST corresponding to the root of the X-BST– it'll first verify that this is not a node that is contain inside the (blue) X-hole and then by a range query in the Y-BST corresponding to the root, it thinks that there are $8 - 3 = 5 > 0$ live points. It will then go to the node corresponding to the x-range $[1, 2]$. Here again it'll first verify that it is not contained in the existing X-hole and then it'll think that this sub-tree has $5 - 2 > 0$ live points. It will then go to the node corresponding to the interval $[1]$. Here it will realize that it has hit the existing (blue) X-hole and will then backtrack to the node with interval $[2]$. It will then verify that it does not fall in the existing X-hole and will verify that the corresponding Y-BST has $2 - 1 > 0$ nodes. It will then traverse the Y-BST, first hitting the pink Y-hole and then moving on to the other leaf in the Y-BST, which corresponds to the point $(2, 3)$. The algorithm will first verify that this does not lie in an X-hole and then will find out that it lies in the Y corresponding to $y = 3$ and hence, will discover the Y-hole marked yellow. The live interval then shrinks to $[2]$ and the algorithm continues.